Journal of Advanced Robotics, Autonomous Systems and Human-Machine Interaction ISSN: 3070-6491

PRIME OPEN ACCESS

Volume 2, Issue 4

Research Article

Date of Submission: 02 Oct, 2026
Date of Acceptance: 02 Nov, 2026
Date of Publication: 13 Nov, 2026

XLOG A CUDA-Native Engine for Neurosymbolic Integration

Levi Dubrovin[1*], Nikita Pospelov and Kirill Sabitov

Levi Dubrovin, Independent Researcher, United States.

Corresponding Author:
Levi Dubrovin, Independent Researcher, United States.

Citation: Dubrovin, L., Pospelov, N., Sabitov, K. (2026). XLOG A CUDA-Native Engine for Neurosymbolic Integration. *J Adv Robot Auton Syst Hum Mach Interact, 2*(4), 01- 30.

Abstract

xlog is a CUDA-native logic programming engine that couples neural perception with deterministic Datalog, probabilistic inference, and epistemic reasoning over world views through a shared typed frontend and provider-owned CUDA runtime. The routes share device data planes but not one universal execution boundary: ordinary Datalog and exact inference are host-orchestrated, while the certified resident recursive and Monte Carlo sampled cores record zero tracked host–device transfers and then return one bounded terminal receipt. End-to-end gradients are realized through the probabilistic path. A CUDA-backed knowledge-compilation pipeline (provenance → CNF → Decision-DNNF → circuit) turns neural outputs into a differentiable circuit whose forward pass is exact weighted model counting and whose backward pass yields exact gradients. The final smoothed circuit is certified against its back-end source formula before it is cached or evaluated: bounded status and error scalars are observed by the host, while resolution-proof traces are checked on the GPU. A resident Monte Carlo alternative covers supported programs for which approximate inference is selected. Circuit caching yields a 2.74× MNIST-addition training speedup, and a worst-case-optimal join subsystem a 27.96× geometric-mean gain over xlog's own binary-join baseline. Against external engines the picture is mixed: 2.8× faster per-epoch MNIST-addition training than Scallop at equal accuracy; on three hub-skewed triangle-counting cases, Soufflé-to-fused-xlog execution-time ratios are 2.37×, 7.95× and 9.00×, with fused peak device allocations of 85–359MB while every materializing arm completes; and exact inference is correctness-equivalent to but slower than ProbLog2. On a public video benchmark, a proximity predicate trained through symbolic credit alone, never shown a distance label, replaces hand-set geometry at no cost in held-out accuracy; the same detector inside the Event-Calculus rule search does not survive ten-fold cross-validation, and on a leak-free split the committed clause does not transfer. On a maritime corpus of thousands of gold events, weighted clauses beat crisp selection by 0.065 F1, and a single chronological training pass reproduces that result exactly.

Introduction

Neurosymbolic systems combine neural perception with symbolic reasoning, but established integrations often place the two halves on different processors. Deep learning frameworks—PyTorch, JAX—execute dense tensor computation on the GPU through optimized CUDA kernels, while systems such as DeepProbLog and NeurASP use CPU-side symbolic backends. Their integration can therefore cross the PCIe bus for network outputs, symbolic results, or gradients. At scale, repeated host–device movement can dominate wall-clock time and bandwidth.The transfer wall is one half of the problem; fragmented interfaces are the other. Existing neurosymbolic frameworks commonly bridge a neural network to one symbolic backend (DeepProbLog to probabilistic logic, NeurASP to answer-set programming). GPU logic efforts, conversely, accelerate focused workloads such as deterministic Datalog or SAT. xlog explores how a typed language and shared CUDA services can support deterministic, probabilistic, epistemic, and neural-symbolic use without pretending that those routes have identical execution or differentiability properties [1-5].

xlog is a CUDA-native logic programming language whose compiler and runtime treat a neural network as an ordinary predicate. Network outputs can become weighted facts for probabilistic knowledge compilation, where exact query gradients flow back to the network. Other reasoning modes reuse the typed relational and solver interfaces without claiming the same gradient semantics. Device-resident buffers carry facts, deltas, circuits, solver state, and gradients,

while the host performs orchestration, I/O, compilation, and bounded control reads. The certified resident recursive and Monte Carlo sampled cores provide the stronger measured boundary: no tracked host–device transfers inside the core, followed by one terminal receipt. Verified knowledge compilation certifies the final smoothed circuit against its back-end formula before use, and DLPack exports device gradient buffers directly into PyTorch's autograd graph [6].

The Principal Contributions of this Paper Are:

- **A Shared Neurosymbolic Integration Model:** A neural network is a predicate, and typed devicebuffer interfaces connect it to the supported reasoning routes. End-to-end gradient training is realized specifically through the probabilistic knowledgecompilation path (Sections 2 and 6).
- **CUDA Relational Execution with a Certified Resident Route:** Semi-naive Datalog evaluation with stratified negation and aggregation uses custom CUDA relational kernels. Eligible recursive query plans are selected automatically for a resident conditional graph whose measured core has no tracked transfers; the wider ordinary executor remains hostorchestrated. A worst-case-optimal join (WCOJ) subsystem avoids the intermediate-relation blowup of binary-join plans on skewed cyclic queries (a measured 27.96× geometric-mean ablation, Section 4).
- **Verified knowledge Compilation:** A CUDAbacked provenance → CNF → Decision-DNNF → circuit pipeline certifies the final smoothed circuit against its back-end source formula before caching or evaluation. The host observes bounded solver status and error scalars; resolution-proof traces are checked on the GPU. Structural invariants required for weighted model counting are established separately by construction (Section 5).
- **End-to-End Differentiable Training with Zerocopy Interop:** Compiled circuits are cached across iterations and evaluated as level-parallel forward/backward kernels, with gradients exported zero-copy via DLPack and Arrow; circuit caching yields a measured 2.74× training speedup. Term embeddings and differentiable ILP ride the same device-resident path (Section 6).
- **Perception Learned Through Symbolic Credit on an External Benchmark:** On the CAVIAR video-event corpus, under a direct per-timestep target, the rule search returns the same theory whether its proximity predicate is the hand-set geometric one that dedicated rule learners are handed or a network trained by the logic's own credit alone, never shown a distance label or any target of its own, at no cost in held-out accuracy. Event-Calculus rules over the precomputed predicate are induced under pre-registered gates; substituting the learned detector inside that search is where the result currently fails (Section 7).
- **Rule Induction at Scale:** On the Brest AIS maritime corpus (3,548 gold rendezVous intervals) the same protocol, over pair-level folds and permutationnull gates, measures in isolation the two mechanisms the published Event-Calculus learners are built on. Weighting the clauses of the same gated pool raises micro F1 by 0.065 over crisp selection, against our own prior expectation that it would not, and learning those weights in a single chronological pass reproduces the batch predictions exactly, for a reason we can name. We draw no comparison with the published maritime figure, whose temporal grid differs from ours (Section 8).
- **Reasoning Beyond Probability:** Epistemic reasoning over world views (know/possible under founded and Gelfond-style semantics) and wellfounded semantics execute on the same substrate over finite supported programs, including positive modal fixpoints and supported recursion through negation evaluated by GPU-backed well-founded semantics, reusing the CDCL, join, and circuit machinery instead of a separate engine (Section 9). The accepted fragment and its bounds are in Section 12.

Target Workloads

Avoiding repeated data-plane copies matters most where the symbolic workload is large and repeated: program-analysis queries over millions of ground atoms, probabilistic logic with many training iterations that share one compiled circuit, and knowledgegraph reasoning with trainable entity embeddings. Small tasks such as MNIST addition exercise the integration end-to-end but sit below this regime. They demonstrate correctness and the circuit-caching effect, with the measured transfer component secondary: approximately 10% of a training step at a standard batch and lost in runto-run variation when a step makes only a handful of neural→symbolic handoffs (Section 10.6). Accordingly, xlog targets NVIDIA-GPU workloads whose byte working set and fixed capacities fit the selected route (Section 12) and that need neural perception coupled to exact, verified probabilistic inference rather than fuzzy relaxations.Section 2 presents the integration model and system architecture. Section 3 describes the xlog language. Section 4 details the GPU-resident symbolic substrate. Section 5 presents verified knowledge compilation, the central technical result. Section 6 ties the pieces together as end-to-end neurosymbolic learning. Section 7 puts that surface on an external video-event benchmark, and Section 8 carries the same induction protocol to a larger maritime corpus. Section 9 extends the substrate to epistemic and other reasoning modes. Section 10 reports evaluation results, Section 11 surveys related work, Section 12 discusses limitations and future work, and Section 13 concludes.

Integration Model and Architecture

The Integration Model

In xlog a neural network is a predicate. A declaration binds a network to a predicate symbol; evaluating that predicate runs a forward pass, and the network's softmax over a label set can become a distribution over weighted facts. Typed device-buffer interfaces make those values available to the supported reasoning routes. Exact gradients of query values flow back through the probabilistic knowledge-compilation route; deterministic and epistemic execution reuse the integration surface without claiming that gradient semantics. Facts, circuit values, and gradient buffers remain device-resident at their data-plane handoffs, while orchestration and bounded control reads remain host responsibilities.

GPU Residency Model
xlog keeps runtime data-plane state in GPU memory and does not silently spill it or switch to a host executor. The provider owns one device handle, stream pool, asynchronous resource, runtime, and memory manager. Exceeding a configured byte budget raises resource_ exhausted; exhausting a fixed row, tuple, sample, or other count capacity raises capacity_exceeded; and a bounded fixpoint that reaches its iteration limit raises convergence_failure. Transfer counters delimit measured claims. In particular, the certified resident recursive and Monte Carlo sampled cores record zero tracked transfers, then return one bounded terminal receipt whose status is authoritative.The payoff is bandwidth asymmetry. PCIe 4.0 ×16 delivers roughly 32GB/s, while GPU HBM provides 1– 3TB/s. A single unnecessary D2H round-trip for a moderate relation (10M tuples at 16 bytes, ≈160MB) costs about 5ms. Keeping fact stores, deltas, circuit values, solver state, and gradient buffers on the GPU avoids copying those data planes through host memory; this claim does not erase the bounded control observations documented for each route.

Crate Architecture
The workspace is organized into five tiers with strict layering: no crate depends on a peer or a higher tier (Figure 1). A leaf core crate defines shared scalar types, error and memory-budget types, canonical boolean parsing, and a bidirectional symbol table for dictionary-encoded strings. Above it, a tier of intermediate representations and device services wraps CUDA via cudarc, resolves embedded kernel modules, and exposes Arrow and DLPack interoperability. Each CUDA provider owns its complete runtime resource graph; there is no process-wide runtime singleton or alternate provider path. The three core subsystems (the typed language frontend of Section 3, the GPU runtime executor, and the GPU CDCL/local-search solver) compose into integrated pipelines for deterministic, probabilistic, and neural-symbolic execution, which a PyO3 extension module (pyxlog) and a command-line binary surface to users [7,8].

From Program to GPU Plan

A Program is Compiled to a GPU-Executable Plan in Five Stages
Parsing: A PEG parser built with pest produces an AST covering the full surface, including probabilistic facts, annotated disjunctions, epistemic literals, and neuralpredicate declarations.
Dependency Analysis: builds a graph with positive, negative, aggregate, probabilistic, and modal edges and computes its strongly connected components (SCCs) with Tarjan's algorithm. It establishes a deterministic stratum order for the ordinary monotone core and classifies supported nonmonotone components for their dedicated plans.
Lowering: Fans out from the normalized AST into three reasoning IRs: deterministic and reduced ordinary rules become relational-IR trees, probabilistic provenance becomes PIR, and modal programs retain their semantics in EIR. Neural predicates compose with the applicable route. SAT/MaxSAT and verification use a separate shared service instead of a fourth reasoning IR.
Optimization: pushes predicates through joins, while lowering chooses a deterministic greedy join order. Shapespecific selectivity rewrites cover recognized triangle and four-cycle plans; general dynamic-programming join ordering is not active. Eligible multiway rules are promoted to the WCOJ subsystem, and bound recursive queries can be rewritten via magic sets (Section 4).
Plan Assembly: emits the route-specific execution plan, whose relational work is dispatched to CUDA kernels. Supported negated recursion uses GPU-backed wellfounded semantics instead of the ordinary semi-naive schedule.

Reasoning IRs and Circuit Format
Three reasoning IRs keep the backends decoupled; probabilistic knowledge compilation also emits a downstream circuit format. Each form has one role.
RIR (RelationalIR) is the algebraic tree (Scan, Filter, Project, Join, GroupBy, Union,Distinct, Diff, Fixpoint, and the worst-case-optimal MultiWayJoin/ChainJoin) carrying cardinality, skew, and delta-vs-full metadata. Deterministic execution and reduced ordinary subprograms consume RIR; the other backends preserve additional semantics in their own IRs.

PIR (Provenance IR) is the weighted Boolean formula (Lit, NegLit, And, Or, Decision) derived from provenance tracking over a probabilistic program, capturing its structure without execution order.

XGCF (xlog GPU Circuit Format) is the com-piled, device-resident form: a levelized DAG whose level offsets let every node at one topological level evaluate in a single kernel launch, with forward (log-space weighted model counting) and backward (adjoint) passes.

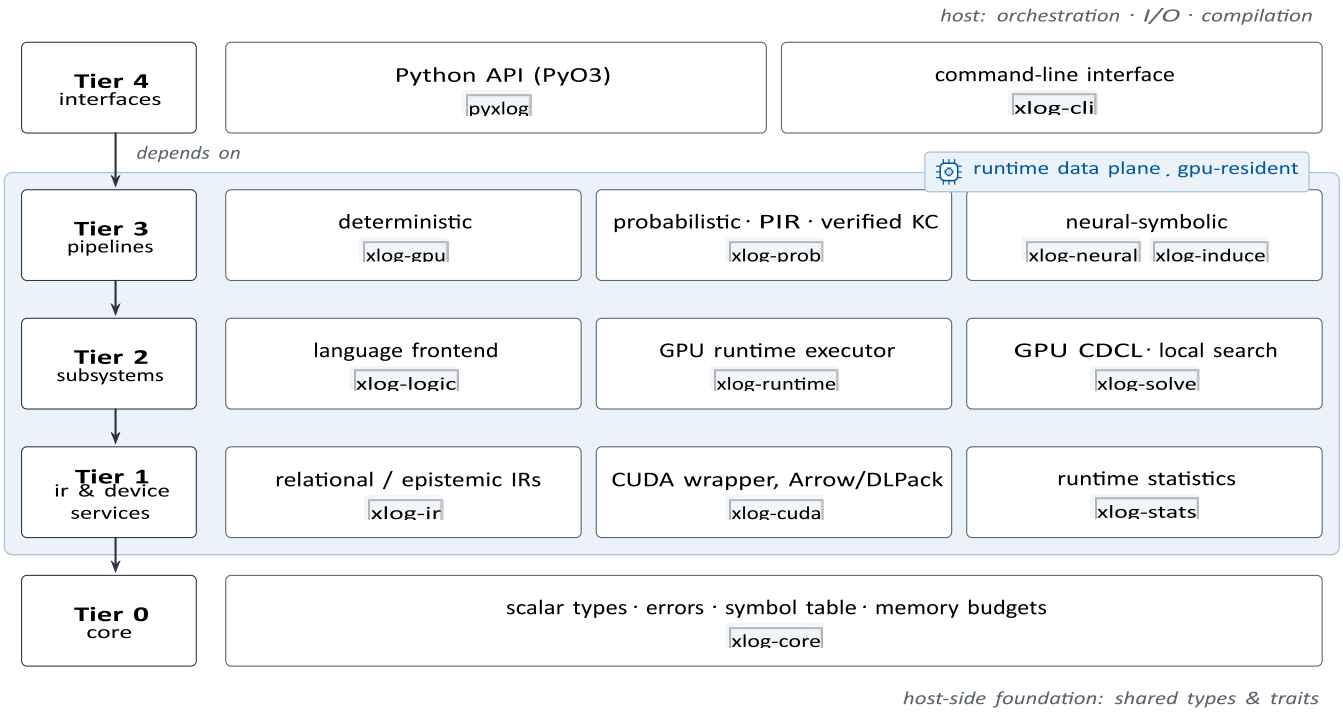


Figure 1: Xlog Crate Hierarchy. Each Tier Depends Only on Lower Tiers; the Shaded Device Plane Marks GPU-Resident Runtime Data Planes, While the Host Provides Orchestration, I/O, Compilation, and Bounded Control Reads

EIR (Epistemic IR) preserves modal operators for world-view reasoning and lowers to a dedicated GPU execution plan instead of ordinary relational algebra (Section 9).

GpuCnf and CDCL form a shared solver service, not a fourth reasoning IR. Callers provide a device-resident CNF formula for SAT/MaxSAT search or verification. The CDCL service consumes that formula directly; XGCF remains the circuit format produced only by probabilistic knowledge compilation.

The X log Language
Probabilistic facts (p::f.) and neural predicate declarations (nn/k) are language-level constructs covered in Sections 5 and 6.

Design Principles
Three principles shape the surface, each chosen to enable GPU execution.
Typed Predicates: every predicate has a statically known signature over a closed set of scalar types (u32, u64, i32, i64, f32, f64, bool, symbol), supplied by a pred declaration or inferred during compilation, so argument mismatches are caught before any kernel is generated and allocation stays bounded and columnar.

Classified Dependencies: negation, aggregation, modal operators, and recursion interact through SCC analysis on the predicate dependency graph. The ordinary deterministic core requires a monotone or stratified schedule, while supported nonmonotone probabilistic and epistemic components enter dedicated plans, including GPU-backed well-founded semantics for supported recursion through negation.

One Language for Many Paradigms: probabilistic facts, annotated disjunctions, neural predicates, and SAT constraints share the syntactic core of deterministic Datalog; parsing, normalization, and dependency analysis are shared, while backend-specific lowering preserves the semantics needed by each plan. Unsupported shapes and unbounded terms are rejected before execution, not routed into an implicit fallback.

A Minimal Program Exhibits the Core Surface:

```
// Typed predicates, facts, a rule, a query. pred
edge(u32, u32). pred reach(u32, u32).

edge(1, 2). edge(2, 3). edge(3, 4).

reach(X, Y) :- edge(X, Y). reach(X, Z) :- reach(X, Y),
edge(Y, Z).

?- reach(1, N).
```

Listing 1: Minimal xlog program: typed predicates, facts, a recursive rule, a query

Type System

Every predicate is assigned a schema over the eight scalar types, with f32/f64 following IEEE 754. A pred declaration supplies that schema explicitly; when it is omitted, compilation infers the schema from facts, rule heads, typed body columns, and arithmetic bindings. String literals ("alice") map to symbol through a global symbol table, giving the runtime dense integer identifiers while preserving source-level readability; domain declarations introduce named aliases. Type consistency is enforced by predicate schemas and lowering-time checks: variable occurrences must agree with known column types, constants are validated against their destination columns, and arithmetic expressions preserve scalar types.

```
pred node(u32, symbol). pred
connected(u32, u32). pred
bridge(u32, u32).

node(1, "alice"). node(2, "bob"). connected(1, 2).

bridge(A, B) :connected(A, B),
    node(A, _), node(B, _).
```

Listing 2: A well-typed xlog program: typed bridge predicate

Declaring bridge so that its head argument is constrained at two incompatible types (e.g. symbol in the head but u32 through connected) rejects the program at compile time.

Expressions, Arithmetic, and User Defined Functions

Rule bodies may contain arithmetic, comparisons, and calls. Arithmetic bindings use the is operator (D is X + Y); the right-hand side supports + - * / %, the builtins abs, min, max, pow, cast, and user-defined functions. Comparisons compile to Filter nodes and push below joins where profitable. Float comparison uses a total ordering, so NaN and signed zeros yield deterministic results instead of contaminating downstream aggregations.

```
pred point(u32, f64, f64). pred
close(u32, u32).

point(1, 0.0, 0.0). point(2, 0.5, 0.5).
point(3, 5.0, 5.0).

close(A, B) :point(A, Xa, Ya),
    point(B, Xb, Yb),
    A < B,
    D is (Xa - Xb) * (Xa - Xb)
        + (Ya - Yb) * (Ya - Yb), D < 1.0.
```

Listing 3: Arithmetic in rule bodies: pairwise proximity via squared Euclidean distance

A user-defined function (UDF) is introduced with func, with optional parameter types and a return type after ->. Arithmetic bodies (optionally with if/then/else) inline into the same Expr trees as builtin arithmetic and participate in the same predicatepushdown and expression-level optimization. A predicate body, as in func get_parent(Child) = Parent :- parent(Child, Parent)., contributes its relational literals immediately before the calling is binding in an ordinary rule or integrity constraint. Its relation may contribute zero, one, or many caller rows; the function syntax does not impose

uniqueness, while ordinary set semantics deduplicates identical projected tuples. Each invocation alpha-renames its non-parameter result and body-local variables, and multiple calls expand from left to right. Nested expansion is depth-bounded; relational calls in conditional result branches and non-term arithmetic arguments are rejected, not hoisted or coerced with changed semantics.

```
pred raw_score(u32, f64). pred
norm_score(u32, f64).

func clamp01(X: f64) -> f64 =
    if X < 0.0 then 0.0 else if X >
    1.0 then 1.0 else X.

raw_score(1, -0.3). raw_score(2, 0.7).
raw_score(3, 1.4).

norm_score(Id, N) :raw_score(Id, R), N is
    clamp01(R).
```

Listing 4: User-defined function: clamp01 normalizes a raw score into the unit interval

Modules and Imports

Programs decompose into .xlog modules on a search path. The use directive loads a module (use graph.) or selectively imports names (use utils/math::{abs, clamp}.); a private modifier hides declarations from importers. The resolver traverses the import graph, detects cycles, validates participating declarations, and performs bounded cross-module schema inference for undeclared predicate contributions before merging them. The resulting logical program then undergoes full compiler type inference, dependency analysis, and route-specific execution planning, so imported rules participate as if written inline.

```
// library: graph_lib.xlog pred edge(u32,
u32). pred reach(u32, u32). edge(1, 2).
edge(2, 3). edge(3, 4).

reach(X, Y) :- edge(X, Y). reach(X, Z) :- reach(X, Y),
edge(Y, Z).
```

Listing 5: Module graph_lib.xlog: predicate declarations and recursive reach rules.

```
// entry: graph_main.xlog use graph_lib.

?- reach(1, N).
```

Listing 6: Entry graph_main.xlog: imports graph_lib and queries reach.

Aggregations and Constraints

Aggregation operators—count, sum, min, max, logsumexp—appear at rule heads with an explicit aggregation variable and lower to GroupBy nodes executed as radix-sort-and-reduce kernels; the stratifier rejects aggregation inside a recursive SCC. Integrity constraints are headless rules (:- body.) asserting that body is unsatisfiable once evaluation reaches fixpoint; a violation raises a diagnostic. logsumexp is included because it is the workhorse aggregation for probabilistic circuits, where log-space numerical stability is essential (Section 5).

```
pred edge(u32, u32). pred
degree(u32, u32).

edge(1, 2). edge(1, 3). edge(1, 4). edge(2,
3). degree(X, count(Y)) :- edge(X, Y).

:- edge(X, _), not degree(X, _).

?- degree(X, N).
```

Listing 7: Stratified aggregation: per-source out-degree with an integrity constraint.

Completeness Over Finite Domains
The surface extends toward language completeness while preserving GPU-native execution: accepted constructs normalize into the typed AST and lower through their semantic backend, while forms that cannot be lowered safely are rejected at compile time, not executed on a hidden CPU interpreter. The accepted extensions are finite typed lists ([], [H|T], the list<T> column type), statically-resolvable meta-predicates (ground, functor, =.., source-fact findall, maplist), explicit stratified negation-as-failure over deterministic relations, magic-set rewriting of bound recursive queries, and finite probabilistic aggregates. Deterministic extensions lower to relational joins and aggregations in RIR; probabilistic aggregates preserve their probabilistic intermediate representation. Open-ended generators, dynamic database mutation, unrestricted call/N, and runtime-variable predicate names remain outside the contract.

GPU-Native Symbolic Substrate
The deterministic Datalog engine supplies CUDA relational operators and device-resident relation storage used across the system. The algorithmic approach—semi-naive fixpoint iteration over stratified programs—is well established; the contribution here is engineering. The ordinary executor is host-orchestrated. For eligible recursive query plans, automatic selection instead uses a certified resident conditional graph: its core runs one recorded launch and synchronization with zero tracked host–device transfers, then returns one pinned terminal receipt carrying status, counters, trace metadata, and schema selection rather than tuple data.

Semi-Naive Evaluation On GPU
The executor processes an execution plan as strata ordered by the dependency analysis of Section 2. Nonrecursive SCCs evaluate each rule once, dispatching its RIR tree to the appropriate kernel and merging results per head predicate. Recursive SCCs run semi-naive iteration (Algorithm 1): each rule is re-evaluated through delta-rewritten variants (one per recursive scan occurrence, so a self-join such as p(X,Y), p(Y,Z) contributes two variants) and the new tuples are unioned, differenced against the full relation, and deduplicated, all on-device, until no predicate produces new tuples. The profiler records per-operator timing, row counts, and peak memory, feeding the optimizer.

Relational kernels
xlog implements its relational algebra as direct CUDA kernels rather than wrappers over cuBLAS, cuSPARSE, **Algorithm 1:** GPU semi-naive evaluation of a recursive SCC. All buffers are device-resident.

Require: SCC rules R; relation store S (device-resident)
Ensure: least fixpoint of R merged into S

for each rule $r \in R$ **do**
  $\Delta_r \leftarrow \text{Eval}(r,S) \setminus S$ ▷ *seed Δ; set difference on device*
repeat
  for each rule $r \in R$, recursive scan i in r **do**
    $T_{r,i} \leftarrow \text{Eval}(r[\Delta \text{ at } i],S)$ ▷ *one Δ-variant per self-join*
  for each head predicate p **do**
    $\Delta_p \leftarrow \text{Dedup } S_{r,i \text{ for } p}\, T_{r,i} \setminus S_p$
    $S_p \leftarrow S_p \cup \Delta_p$ ▷ *union + sorted dedup on device*
until all $\Delta_p = \emptyset$ **or** iteration cap reached
free all Δ buffers

or Thrust. Bespoke kernels make allocation, synchronization, and host staging explicit and keep relational intermediate buffers on-device. Table 1 summarizes them. Table 1: Core relational CUDA kernels.

Kernel	Purpose
join	Hash join with linked-list collision chains and FNV-1a composite hashing over raw key bytes, so all scalar types share one path; count→scan→materialize variants for inner and left-outer joins.
sort	Stable 4-bit radix sort (8 passes over 32-bit keys): per-pass histogram, prefix sum, scatter.
filter	Comparison operators for column-vs-constant and column-vs-column, with stream compaction via prefix sum.
groupby	Sorted-input grouped aggregation: boundary detection, then per-group Count/Sum/Min/Max/LogSumExp reduction.
dedup	Sort-based deduplication and set difference with type-aware equality, including IEEE 754 float handling.
set_ops	Union (concat+sort+dedup) and set difference via sorted-array binary search.
scan	Blelloch parallel prefix sum, the building block for filter compaction, dedup, and group-by.
pack	Multi-column key packing into row-major byte arrays with FNV-1a hashing.

The filter kernel makes a correctness-critical choice for floating point. Equality uses standard IEEE 754 semantics (NaN ≠ NaN); ordered comparisons use a total-ordering transform that maps an f64 bit pattern to an i64 whose integer order matches the IEEE 754 totalOrder predicate (−NaN < −Inf < negatives < −0.0 < +0.0 < positives < +Inf < +NaN), so Datalog programs produce deterministic results for all floatingpoint inputs.

Adaptive Join Planning
The optimizer performs predicate pushdown using runtime statistics, decomposing filters above joins into left-, right-, and cross-predicates. Lowering chooses join order with a deterministic greedy heuristic. A selectivity pass recognizes triangle and four-cycle shapes, and prior execution statistics can be exported and merged before a later compilation. A general dynamic-programming join ordering is not active; configuration fields that resemble such a threshold do not change this planning path.

Worst-Case-Optimal Joins
Binary-join plans can materialize intermediates asymptotically larger than the final result, the classic pathology for cyclic queries such as triangle enumeration. This is not a corner case for xlog's target workloads: program-analysis queries (points-to, call-graph and reachability inference over code-dependency graphs) can contain cyclic, skewed joins where intermediate size governs cost. xlog addresses this with a worst-case-optimal join (WCOJ) subsystem. A pure-Rust hypergraph planner analyzes rule bodies for multiway-join eligibility, derives a deterministic variable ordering from cardinality and selectivity statistics, and emits a MultiWayJoin/ChainJoin node when promotion preserves output semantics; other rules retain their ordinary binary-join plan.The GPU kernel executes the join via count→device prefix scan→materialize over flat sorted-column storage, with a four-byte metadata D2H total and no countvector transfer. Coverage spans the triangle, 4-cycles, K-clique templates (K=5–8), recursive/SCC integration, and helper-relation splitting that exposes inner skew to root-level histograms. A default-on skew classifier decides per query whether to dispatch WCOJ: high-skew inputs take the WCOJ path, while uniform or empty inputs select the binary-join chain. Section 10.2 reports the measured speedup [9-11].

Runtime Optimization
Interactive and long-running deployments (REPL sessions, iterative solver loops, incremental training) repeatedly evaluate similar queries over relations that change little between calls. A stateless executor rebuilds join indices and recomputes shared subplans each time, paying the dominant cost repeatedly. Three optimizations exploit this between-call stability, each with explicit controls and zero added data-plane transfers. Commonsubexpression elimination caches safe deterministic subplans within an evaluation. Adaptive re-optimization adopts a compiler-supplied candidate plan when misplan telemetry crosses a configurable ratio, with outputequivalence checks and rollback. And a persistent hashindex manager, keyed on relation generation, schema, and device, retains built indices across repeated sessions with deterministic LRU eviction (Section 10).

Reversible Symbols
Datalog programs operate on strings, but GPU kernels operate on fixed-width numeric columns. xlog interns each unique string to a dense, sequential u32 identifier and resolves it back in O(1) at output time. Symbol columns then are ordinary u32 columns (join, sort, filter, and dedup kernels operate on them with no special-casing) and the string table stays host-side, since variable-length data is needed only at ingestion and output. The symbol type maps to Arrow's Dictionary(UInt32, Utf8), so export to columnar consumers (cuDF, Polars, DuckDB) is zero-copy while preserving the compact internal representation.

Verified Knowledge Compilation
Knowledge compilation is the route that makes a neural network and a probabilistic logic program differentiable end-to-end. A network's softmax becomes a distribution over weighted facts; provenance tracking yields a weighted Boolean formula; and that formula compiles into an arithmetic circuit whose forward pass is exact weighted model counting and whose backward pass yields exact gradients. Probabilistic systems such as ProbLog follow this compile-once/evaluate-many model on the CPU. xlog instead keeps the compilation and evaluation data planes in CUDA buffers while the host orchestrates stages and reads bounded control state. Before use, xlog certifies the final smoothed circuit against its back-end source formula; this certificate does not cover provenance extraction or CNF encoding [12].

The Compilation Pipeline
The pipeline transforms a probabilistic program into a GPU-resident arithmetic circuit (the XGCF format) through five host-orchestrated stages (Figure 2). Their intermediate data stays in device buffers, while solver completion is observed through bounded status and error scalars.

- **Provenance Extraction:** Provenance over deterministic evaluation produces a weighted Boolean formula. An on-device interner hash-conses node batches through an atomic GPU hash table, deduplicating structurally identical subformulas; this is essential because grounding can produce exponentially many redundant subformulas, and interning on the host would force the uninterned graph into host memory—reintroducing the very transfer the pipeline exists to avoid [13].
- **CNF Encoding:** A Tseitin transformation lowers the formula to CNF on the GPU, using parallel prefix sums to assign compact, gap-free variable IDs and emit clauses into a device-resident compressed-sparse-row structure [14].

- **D4 Compilation:** The CNF compiles into a DecisionDNNF (d-DNNF) circuit via a GPU-native variant of the D4 algorithm : a BFS frontier exposes parallelism, per-block DFS workers perform component detection, decision-variable selection by VSIDS-like activity, and recursive decomposition. A smoothing pass forces every branch of an OR/decision node to mention the same variable set, and the result is levelized so each topological level evaluates in one kernel launch. Decomposability, determinism, and smoothness (the structural properties that make weighted model counting over a Decision-DNNF correct are established by construction. Certification runs after smoothing and therefore gates the exact final circuit that is cached and evaluated [15-19].

Verification: a Machine-Checked Certificate

xlog does not trust the final compilation stage; it proves the final smoothed circuit C equivalent to its back-end source formula φ (Algorithm 2). It solves φ ∧ ¬C and C ∧ ¬φ on a GPU CDCL solver, and both must be unsatisfiable for C ≡ φ. UNSAT results carry resolution proofs checked on-device. The host reads bounded status and error scalars before accepting that GPU proof result; a wrong, incomplete, or budget-limited result returns a typed error and never reaches circuit caching or evaluation. The certificate says that the circuit computes the same Boolean function as φ; the count-correctness invariants are guaranteed separately, by construction. The compilation back-end thus becomes a machine-checked stage instead of an assumption, closing the failure mode in which a silently miscompiled circuit would corrupt every downstream probability and gradient. The check covers the back-end stage specifically: provenance extraction and CNF encoding are trusted, not certified. CPU-side certified knowledge compilation exists ; the contribution here is performing the check on-device, sharing the GPU substrate. The same device-resident CDCL substrate is reused for SAT/MaxSAT queries and for epistemic candidate validation (Section 9) [20].

Algorithm 2: Verified knowledge compilation. The final smoothed circuit is accepted only if proven equivalent to its back-end source formula.

Require: weighted Boolean formula φ (device-resident)
Ensure: verified circuit C (XGCF), or typed error

$\varphi \leftarrow$ InternHashCons(φ) ▷ *dedup via atomic GPU hash table*
$\Phi \leftarrow$ TseitinCNF(φ) ▷ *prefix-sum variable IDs; CSR clauses* $C \leftarrow$ D4(Φ); $C \leftarrow$ SmoothLevelize(C)
$u_1 \leftarrow$ CDCL($\varphi \wedge \neg C$); $u_2 \leftarrow$ CDCL($C \wedge \neg\varphi$)
observe bounded status and error scalars for u_1, u_2 on host
if $u_1 \not\in$ Unsat **or** $u_2 \not\in$ Unsat **then**
  return typed verification error
check resolution proofs of u_1, u_2 on-device
return C ▷ *cache the exact circuit that was certified*

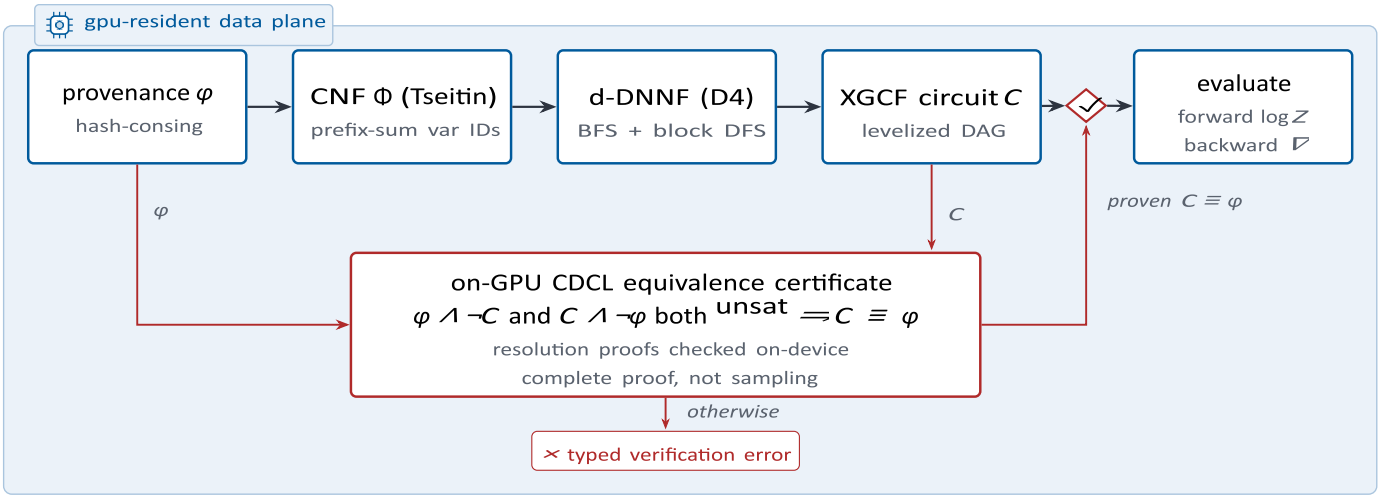


Figure 2: Verified Knowledge-Compilation Pipeline. The Data Plane Uses CUDA kernels and Device-Resident State Under Host Orchestration. The Final Smoothed Circuit C Reaches Evaluation Only Through the Equivalence Gate (✓): An on-GPU CDCL Solver Proves φ ∧ ¬C and C ∧ ¬φ Both Unsatisfiable; the Host Observes Bounded Status and Error Scalars, and Any Failed or Incomplete Proof Returns a Typed Error

The verified circuit supports a level-parallel forward pass (log-space weighted model counting, one kernel per level, yielding logZ) and a reverse-order backward pass that accumulates per-variable gradients. Both leave their results in GPU-resident buffers consumable directly by PyTorch via DLPack (Section 6).

Circuit Caching
Building and certifying a circuit is the most expensive stage, on the order of a minute for a cold start on the MNIST-addition benchmark, and a naive implementation pays it every epoch. Measured separately, it is the equivalence proof and not the D4 compile that dominates that time (Section 10.6). The key observation is that the circuit structure depends on the propositional structure of the grounded program and the evidence pattern, but not on the numerical weights. When neural parameters change between iterations, the weights on probabilistic facts change while the Boolean formula and its compiled circuit stay identical. The certified final smoothed circuit is therefore cached, keyed by a structural hash of the provenance graph and evidence configuration; subsequent iterations update only the weight buffers and run the forward/backward kernels. On MNIST addition that amortization is what the circuit-cache ablation of Section 10.3 measures.

Monte Carlo Alternative
When the user selects approximate inference, xlog provides a GPU-parallel Monte Carlo engine for its supported fragment. It draws values for probabilistic facts on the GPU and evaluates the deterministic program in each sampled world, using rejection sampling or, when every evidence atom is a Bernoulli fact, evidence clamping that forces observed values and eliminates rejection waste. Both methods report confidence intervals. The resident sampler evaluates all worlds in a single device launch with a bounded device-side fixpoint. Its sampled core records zero tracked transfers; after synchronization, one pinned receipt returns status, counts, and trace metadata. That receipt's convergence status is authoritative: reaching the iteration bound returns convergence failure, even if partial counters exist. Fixed-count exhaustion returns capacity exceeded, byte-budget exhaustion returns resource exhausted, and unsupported fragments return their typed diagnostic; none selects a host execution path.

Neurosymbolic Integration
This section is where the relational substrate (Section 4), verified probabilistic knowledge compilation (Section 5), and device-buffer interoperability meet. Network outputs feed weighted facts into the probabilistic route; the final smoothed circuit is certified before use; and DLPack carries device gradient buffers into the neural optimizer without copying them through host memory. Host orchestration and bounded control reads remain outside that zero-copy data-plane handoff.

Neural predicates
In xlog a neural network is a predicate. The nn/4 declaration embeds a network directly into the program:

```
nn(network_name, [input_vars], output_var, [output_labels]) :: predicate(args).
```

Listing 8: Neural predicate declaration (nn/4)

This is not a foreign-function escape hatch: it states that evaluating predicate(args) requires a forward pass through the named network, whose softmax over the label set becomes a distribution over probabilistic facts feeding the knowledge-compilation pipeline of Section 5. On the Python side a PyTorch module is registered against the predicate:

```
program.register_network("mnist_net", net, optimizer)
```

Listing 9: Registering a Py Torch module against a neural predicate.

Registration is typed, not by name. A caller may state the arity of the network's predicate, and that claim is checked against every nn/4 declaration bound to the network in the program and rejected on mismatch: the program's own declaration is the authority. Per-argument catalog sort identifiers may accompany the arity (one per declared argument, refused without an arity or at the wrong length) and an opaque content hash may identify the registered network, so a retrained one mints a new identity when it is registered again. The engine interprets neither; both are carried for consumers, which read them back through a queryable metadata view together with each declaration's predicate, predicate arity, input arity, and label set. A downstream rule learner can therefore check a candidate against what the program declares instead of against a naming convention—the property Section 6.5 relies on.The dataflow runs in two directions.

Forward: when evaluation reaches an nn/4-backed predicate, the registered module runs, and its softmax vector is decomposed into weighted probabilistic facts, one per label, and fed into the knowledge-compilation pipeline of Section 5, whose circuit cache carries across iterations.

Backward: after the circuit computes the forward log-probability and backward gradients via level-parallel kernels, the per-leaf gradient buffers are exported as DLPack tensors and injected into PyTorch's autograd graph, and the optimizer updates the network as usual. The GIL is released during GPU work, so CUDA execution does not block the interpreter. A neural predicate is not confined to a query head: it may also appear inside rule bodies, so that whole rules, not just a single perceptual predicate, become the trainable object (Section 6.4).

End-to-End Training
The canonical example is MNIST addition (the DeepProbLog benchmark): given two digit images, predict their sum, with supervision only on sums: the network never sees individual digit labels.

The Entire Reasoning Structure is Two Rules:

```
nn(mnist_net, [X], Y, [0,1,2,3,4,5,6,7,8,9])
      :: digit(X, Y). addition(X, Y, Z) :-
digit(X, D1), digit(Y, D2), Z is D1 + D2.
```

Listing 10: MNIST addition: a CNN-backed digit predicate and an addition rule

The nn/4 declaration connects the CNN to digit/2; the addition rule computes every consistent decomposition of a sum through probabilistic marginalization. Each query addition(i, j, s) trains by minimizing −logP(addition(i,j,s)) under the compiled circuit. The driver compiles the program, registers the network, and runs the loop, which batches queries sharing a circuit template and runs forward/backward over the cached circuit:

```
program = pyxlog.Program.compile(SRC) program.register_network("mnist_net", net, optimizer) program.add_tensor_source("train",
train_images) history = pyxlog.train_model_tensor(
program, queries, epochs=epochs, batch_size=batch_size)
```

Listing 11: Python driver: compile, register, train.

The decisive performance property (Figure 3) is that compilation and verification happen once, on the first forward pass. Every later iteration reuses the cached circuit, executing only the weight update and the levelparallel forward/backward kernels. This is the mechanism behind the 2.74× training speedup; the loss reduction across seeds confirms that gradients flow from circuit evaluation through the logic program back to the CNN, and the full timing breakdown is reported in Section 10.

Zero-Copy Interoperability
A GPU-native engine is only useful if results reach the frameworks researchers already use, without copies that would reintroduce the transfer wall. xlog exposes its device-resident state through four standard interfaces.

DLPack: Each relation column or gradient buffer becomes a managed DLPack tensor whose GPU memory is reference-counted and freed only when every consumer releases it; on import, xlog validates device, dtype, contiguity, and alignment, optionally against an expected schema. In Python the protocol surfaces as a capsule, so torch.from_dlpack yields a live CUDA tensor backed by the very memory xlog wrote.

Arrow: For columnar tools (Polars, DuckDB, cuDF), relations export through the Arrow C Device Data Interface; symbol columns export as Arrow dictionary arrays, so a consumer reads symbolic columns as native string dictionaries while xlog keeps the compact integer representation [21].

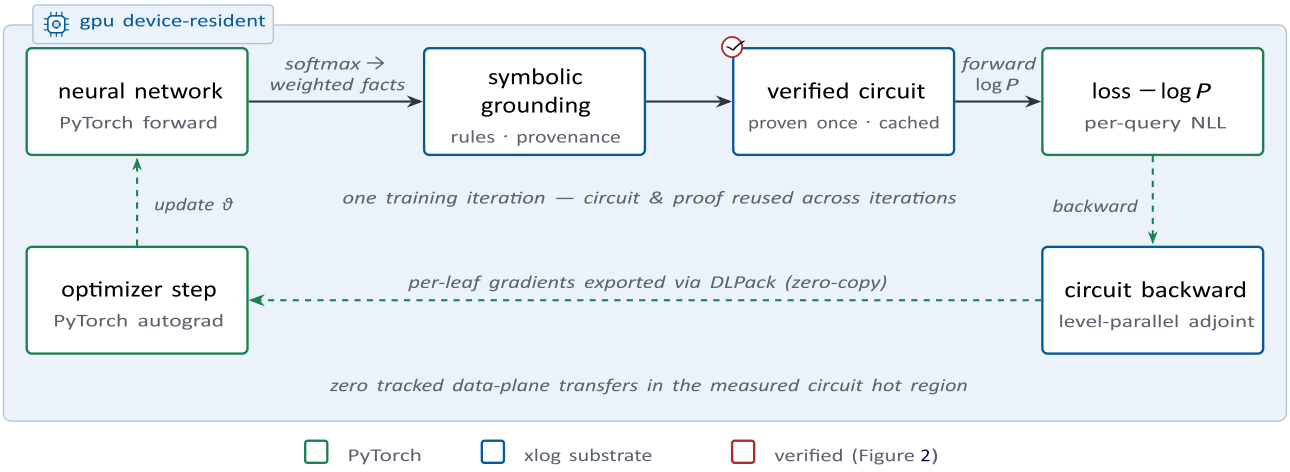


Figure 3: One Neurosymbolic Training Iteration. Neural Tensors, Circuit Values, and Gradient Buffers Occupy One CUDA Device (Shaded Plane); the Measured Circuit Hot Region Records Zero Tracked Data-Plane Transfers. The Host Still Orchestrates Stages and Observes Bounded Control State. The Final Smoothed Circuit is Compiled and Certified Once (✓, Figure 2), then Reused Across Iterations, and Gradients Re-enter PyTorch's Autograd Graph Zero-Copy Through DLPack

Python bindings: The pyxlog extension (PyO3 [8]) exposes compilation, evaluation, training, and result extraction; tensor-valued results are returned as DLPack capsules, and long-running GPU operations release the GIL so data-loader threads proceed concurrently.

PyTorch autograd: The circuit behaves as a differentiable function inside PyTorch: the network's gradenabled forward feeds the circuit, the scalar loss is exported via DLPack and re-imported, and a batched path stacks inputs into one forward pass and runs a single backward, so gradients flow from the logic layer back to nn.Module parameters with no custom autograd Function.

Trainable Rule Bodies

Placing a neural predicate in a rule body lifts xlog from learning a single perceptual predicate to learning whole rules. The tensor and circuit data planes remain deviceresident; route-specific orchestration and control observations are documented separately.

- **Mixed Bodies:** A trainable body may interleave neural-evaluated atoms with ordinary symbolic literals. The symbolic literals act as hard join conditions: they gate which groundings can fire but carry no probability mass or gradient. The head probability therefore factors as a 0/1 indicator of the symbolic condition times the neural probability scaled by a learned per-rule weight $\sigma(w)$, and gradients flow only through the network outputs and the rule weight, never through the ground facts.
- **Existential Joins:** When an existential (non-head) body variable is a neural predicate's input, xlog grounds that predicate over the *real* join domain *inside* the circuit instead of stripping the join as a pre-filter: the neural occurrence expands into one circuit leaf per domain event, and provenance OR-aggregates $\exists e.\ \text{neural}(e) \wedge \text{join}(e,\textit{head})$ at each head binding. This makes "does supporting evidence exist?" rules learnable, reasoning over a whole domain of events instead of only per-instance classification.
- **Joint Multi-Rule Mixtures:** Several same-head trainable clauses combine as a noisy-OR mixture, $P(\textit{headi}) = 1 - Q_k\ 1 - r_k[i]m_k[i]\sigma(w_k)$. Here $r_k[i]$ is a candidate's relational eligibility, $\sigma(w_k)$ its learned rule weight, and $m_k[i]$ an optional graded confidence in carried per binding, for example a fact's confidence trajectory over time. Rule learning thus moves from binary rule eligibility to a continuous, differentiable weighting of competing and cooperating rules.
- **Learned Body Gates:** Inside that mixture a candidate may also carry a neural conjunct over a perbinding entity feature $\phi(x)$: its eligibility becomes its relational grounding *and* a straight-through-thresholded gate $g_\theta(\phi(x)) \geq \tau$. *A* gated clause then competes against ungated ones on the same noisy-OR terms, and g_θ trains jointly with the rule weights under the same held-out selector. By default ϕ is detached where it enters the engine: gradient reaches θ and the rule weight but stops at the neural head, leaving the feature producer frozen. Training the producer too is an explicit opt-in per candidate: one flag leaves ϕ attached, so gradient flows on into the backbone that computed it. Only the autograd linkage changes; the values uploaded are identical either way.

These trainable-rule paths share the verified circuit and the zero-copy substrate of the simpler predicate case, and extend to recursive programs now that provenance converges on two-sided recursive strongly-connected components. Section 7 carries the surface onto a public benchmark, not a demonstration: the perceptual predicate inside an induced Event-Calculus rule body is trained through symbolic credit alone, cross-validated under preregistered gates. Section 12 states what that evidence does and does not support.

Engine-Mode Credit and Candidate Selection

Which of the candidate bodies the engine enumerates should the learner keep, and on what evidence? Learning a rule body is not only a gradient problem: it is also a decision about which candidate to keep, and the two need different evidence. Engine-mode training separates them. The engine already enumerates the candidate space (the well-formed pairs of body relations over the program's own binary relations) and already holds the join extensions those candidates read, so training scores that space directly instead of a reconstruction of it. A candidate's score on a fact is a noisy-OR over the witnesses the engine reports for that fact: a neural body relation contributes a probability per witness, read at the fact's own label, while a purely relational one contributes a fixed {0,1} cover. Candidates combine as a softmax over their logits, and the resulting negative log-likelihood is a single autograd graph, so one backward pass updates the candidate logits and the network behind the probabilities together.

- **A Worked Instance:** Take one head fact and two candidates for its body, one relational and one neural. The relational candidate covers the fact or does not, so its score there is 1 or 0. The neural candidate's score is the noisy-OR over the witnesses the engine reports for that fact, each read at the fact's own label: two witnesses at probability 0.5 give $1-0.5\cdot0.5 = 0.75$, above a relational candidate that does not cover the fact and below one that does. Those are training scores, and the arbiter never reads them. It rescores both on the held-out facts by the same witness semantics, drops either whose held-out score falls below the fit gate, and keeps the relational one when the two land within the score quantum; if that narrowing leaves two relational candidates the data cannot separate, it abstains rather than pick one. The candidate pool is validated against the typed registry of Section 6.1 instead of being trusted by name: a neural relation whose declared arity the credit has no witness semantics for is refused at registration, and a pool that filtering empties reports its per-filter counts instead of silently shrinking. And because raw engine constants index the caller's feature rows directly, that identity is accepted only when the witness domain is exactly the dense row range; a misalignment that would stay in bounds while gathering another event's probability is refused, not guessed at.

- **Selection On Held-Out Generalization:** Training credit cannot distinguish a crisp but coincidental rule from a soft but correct one, even in principle; generalization can. Selection therefore never reads the trained weight. The facts are split into folds; each fold trains on the rest and rescores every candidate on the held-out facts by its own witness semantics; the arbiter ranks the fold-averaged scores. A fit *gate* then drops any candidate whose held-out score falls below a threshold, since a rule that cannot fit held-out data is not a rule. Ties within the score quantum break toward the relational candidate on Occam grounds, but only when that narrowing leaves exactly one: preferring a relational explanation over a neural one is licensed, choosing among relational duplicates the data cannot separate is not. When neither step is decisive the arbiter returns no rule and says why; abstention is an outcome of the selector, not a failure of it. The held-out score may be accuracy or F1, and the choice matters: at a rare positive class accuracy sits on the all-negative base-rate plateau, where a candidate that predicts nothing outranks the best real detector, which is why the benchmark protocol of Section 7 selects on held-out F1 instead.
- **Masked Witnesses and Withheld Evidence:** A perwitness mask lets evidence be withheld rather than denied: a masked witness leaves the candidate's index entirely, contributing zero credit and zero gradient instead of a coerced false, and the facts it thereby leaves uncertain are excluded from the held-out score and counted against a coverage gate that runs before the fit gate, so no candidate is killed by facts it was never allowed to see. The channel rides the accuracy axis only and is refused up front beside the F1 holdout, whose measurability derives from raw labels a mask can hide.
- **Acceptance Without Retraining:** The same machinery also runs as an acceptance instrument: given an externally trained detector, a frozen entry point scores every enumerated candidate against it with no optimizer, no gradient and no folds and applies exactly the arbiter's gates, so where the cross-validated path asks whether a detector can be trained for a rule, this one asks whether a rule is right given the detector one already has. A training-mode module is refused rather than quietly switched, since batch-norm statistics and dropout would de-determinize the bit-identical scores the entry point exists to provide.

Term Embeddings and Differentiable ILP

- **Term Embeddings:** attach dense, optionally trainable vectors to logical symbols. Where nn/4 maps perception to discrete facts, embeddings give logical entities a continuous representation that participates in neural computation, with gradients flowing through the embedding lookup: knowledge-graph entity representations, for instance, trained jointly with neural perception.
- **Differentiable ILP:** learns first-order rules instead of network weights. Candidate rules are represented as sparse bitmasks over the ground-atom space, credit assignment runs entirely on-device via scatter–gather, and a promotion pipeline decides which rules enter the program.Four kinds of judgement sit on these two paths, and Table 2 separates them. Convergence and the supply of explicit held-out examples are preconditions checked before any gate runs, not gates: with neither held-out positives nor held-out negatives supplied, the pipeline returns for manual review before a gate is evaluated, so the always-run gates cannot promote on their own.The benchmark runs of Section 7 exercise the last row only. Its permutation-null threshold is not an extra gate but the holdout arbiter's own fit threshold, derived per fold from label permutations instead of fixed at a constant; the promotion gates above it belong to the differentiable-ILP path and are not on that route.

Judgement	What it judges	Path
Six always-run gates	the rule derives every training positive	dILP and no training negative, passes the novel-fact audit, loses no protected fact, meets a held-out F1 threshold, and names only relations with typed schemas
Two held-out gates	the same rule on the supplied held-out	dILP positives and negatives
Ambiguity scan	which other candidates also explain	the dILP data; reported, never decisive
Permutation-	whether a candidate generalizes better	arbiter null fit threshold than shuffled labels do on its own fold

Table 2: What Judges what, on Which Path: the Promotion Pipeline of Differentiable ILP or the Holdout Arbiter of Section 6.5. Convergence and Supplied Held-out Examples are Preconditions, not Gates: Without them the Pipeline Returns for Manual Review Before any Row Below Runs

Both paths ride the device-resident bridge of Section 6.3. Where the trainable rule bodies of Section 6.4 learn rule weights and per-binding confidences, differentiable ILP learns rule structure; the two are complementary. This sparse, GPU-resident formulation avoids the cubic blowup of dense rule materialization. Unlike the trainable-body surface, this path is not exercised on the benchmark of Section 7, and it is a beta subsystem (Section 12).

Learning Event Rules From Perception

The preceding sections build the machinery; this one puts it on an external benchmark. The task is symbolic event-rule induction over video surveillance data, and the perceptual predicate the induced rules depend on is itself learned through the logic credit alone, with no perceptual supervision anywhere in the loop.The Event Calculus is the standard

logical formalism for narrative reasoning over time: a fluent is a timevarying property of the world, for instance that two tracked people are meeting, and the two event predicates initiatedAt(F,T) and terminated At(F,T) say when something at time T makes fluent F begin or makes it stop. A domain-independent axiom of inertia closes the loop, so that a fluent holds at a time if it was initiated earlier and not terminated since. Reasoning is therefore cheap and generic once the initiation and termination rules are known; the whole difficulty is obtaining them.Those rules are classically hand-written by a domain expert. The line of systems that learns them instead, inductive logic programming over Event-Calculus theories, learns them from a symbolic input vocabulary that somebody else has already computed. On the CAVIAR video benchmark the learner is handed a relation such as close(P1,P2,T), already reduced from raw tracker coordinates by a hand-chosen distance threshold, alongside per-person activity states. Rule induction thus starts one layer above perception, and the perceptual layer's quality is a fixed input, not something the learning explains.

In the runs below the proximity predicate is not given. It is close_nn, a small network over the raw pair coordinates, and it is never shown a distance label, a threshold, or any target of its own. Its only gradient arrives through the induced rules: a candidate clause containing close_ nn in its body is scored by how well the resulting EventCalculus theory explains the frame-level annotations, and that score is what propagates back to the network's weights.This is the trainable rule bodies of Section 6.4 on a real benchmark instead of a demonstration: the neural atom sits inside a rule body, and the symbolic literals beside it act as hard join conditions that gate which groundings fire without carrying gradient. These runs take the engine-mode credit path of Section 6.5: the rule learner's own negative log-likelihood is the network's loss, so structure search and weight training are one differentiable object rather than two pipelines in sequence. The compiled route of Sections 6.3 and 6.4, with its machinechecked equivalence certificate and its zero-copy DLPack export, is the one the MNIST-addition measurements of Section 10 exercise, not the one that trains close_nn here.

Protocol

Two targets are searched throughout this section. The direct target asks, per timestep, whether the fluent holds; the Event-Calculus target asks when it is initiated and terminated, and infers holding through inertia. They carry different class balances, and the guarantees below are stated per target. Except where a table says otherwise, F1 is micro-averaged over pooled per-fold counts, as the published evaluations compute it. Each guarantee below was fixed before the run it governs and covers that run, not the history behind it: the corpus was already in hand while the gates were being designed.

- **Leak-Free Scene-Family Splitting:** The distributed caviar corpus used by the published Event-Calculus learners is a dump of 26 video segments, and it is not clean. A frame-level audit of every meeting transition event in the dump against the original ground-truth XML classifies 21 of 25 events as real, 3 as duplicates and 1 as a splice artifact. Two source videos appear in both the dump's train and its test file, and one video alone contributes 855 duplicated gold pair-frames. Crossvalidating over segments therefore permits same-scene transfer. The clean protocol folds over scene families recovered from the source XML instead, so no scene appears on both sides of a split.
- **Recall-Aware Holdout:** Scoring candidates by heldout accuracy is useless at this class balance (roughly ten positive transitions against tens of thousands of rows), because every candidate sits on the all-false base-rate plateau, and the ranking provably inverts: the best real detector scores below the empty theory. Candidate selection in the Event-Calculus searches uses per-fold held-out F1 instead. The direct per-timestep target does not have this problem, since it carries thousands of positives per fold and not a handful of transition events, so its searches are left at their defaults, held-out accuracy with a fixed 0.75 fit gate. This guard and the next therefore govern the Event-Calculus route only.
- **Permutation-Null Fit Gate:** A candidate must beat what label shuffling alone achieves on its own fold. Each fold derives its own threshold from 1000 label permutations (seed 7) as the 95th percentile of the pool-maximum mean per-fold F1, computed per target and on the training side only. A theory whose clauses do not clear that bar is not returned: the search abstains and reports an empty theory. Abstention is a first-class outcome of this protocol, not a failure mode of it.
- **Two Fold Structures:** The direct per-timestep runs reuse the three folds the distributed dataset file ships, whose datapoints are fixed-length windows of consecutive frames for one person pair (the windowed folds); the Event-Calculus runs fold over whole video segments, or over scene families under the clean protocol.

Results

Perception Learned Through Logic Credit

On the windowed-fold protocol with a direct per-timestep target, the search finds the same two-clause theory on every fold in both modes (Listing 12):

```
ldsAt_meeting(PP,T) :- both_inactive(PP,T), close*(PP,T). holdsAt_meeting(PP,T) :- both_active(PP,T), close*(PP,T).
```

Listing 12: The two-clause theory the search returns on every fold, in both the relational and the neural mode

where close* is the precomputed geometric predicate in the relational mode and the learned close_nn in the neural mode. Held-out F1 per fold is 0.9215/0.6804/0.7695 with the precomputed predicate (mean 0.7905) and 0.9215/0.7918/0.7695 with the learned one (mean 0.8276). The learned detector matches the precomputed predicate's result exactly on two folds and outperforms it on the third, where a soft learned boundary survives the train/test geometry shift better than a hard threshold. That is this section's main positive result: a perceptual predicate trained through nothing but symbolic credit stands in for hand-set geometry without loss, and rests on no published comparison.

Protocol-Matched Cross-Validation

Under the full Event-Calculus target, the combined 26-segment corpus (32,360 co-visible pair-frames, 1,833 gold meeting frames) was cross-validated ten-fold by video segment with every gate re-derived inside each fold. The result is precision 0.6580, recall 0.8271, F1 0.7329. The same initiation clause, both_active & close, is selected on all ten folds, each time over that fold's own permutation null; the termination theory is empty on all ten, so the fluent persists by inertia to the end of each co-visible pair-run.

Published Figures on the Same Corpus

This protocol matches the published Event-Calculus learners on the axis that matters most for the metric: F1 is computed on holdsAt inferred through inertia, on the same distributed corpus, with the same micro-aggregation. The published figures for meeting on that corpus are 0.735 for hand-crafted rules, 0.782 and 0.792 for OLED (the latter as reported by its authors, the former as re-run in the WOLED paper and 0.887 for WOLED-ASP; they are tabulated beside ours in Section 10. Ours sits just under that group's lowest figure. We draw no ordering from any of these differences, in either direction. Five protocol deltas separate the rows and they do not all point the same way: the fold structure, the scope of the learned theory, the body-length limit and the way the reported settings were chosen all differ, and the shared metric is the only reason the comparison is drawn at all. Table 11 prints the five deltas in full beside the figures [22,23].

A Second Vocabulary Iteration

Two pair-level transition relations were added to the Event-Calculus vocabulary, one for a pair crossing the proximity threshold outward and one for strictly growing distance, and the identical folds, seeds and gates were re-run. The result rises to precision 0.7357, recall 0.8260, F1 0.7782 (1514 true positives, 544 false positives, 319 false negatives), with a termination theory now learned on seven of the ten folds. That figure falls inside the 0.735–0.887 span the published systems occupy, but it must never be read as the headline: its vocabulary was chosen *after* the first run's test-side failure mode was known, on the same folds. Everything else about it is pre-registered; the vocabulary is not. Against published estimates selected as the best of several tried, the smaller number declared in advance is the stronger scientific claim, which is why it leads here and 0.7782 appears only with this label attached.

The Learned Detector Under the Event-Calculus Target

Substituting the learned close_nn detector for the precomputed proximity predicate inside the Event-Calculus initiation search does not survive crossvalidation: precision 0.1250, recall 0.0005, F1 0.0011 over the same ten folds (Table 12 gives the counts). Only one fold commits a clause, the same both_active & close_nn shape the geometric predicate selects there, and probed directly against held-out ground-truth proximity that clause scores precision 1.0 at recall 0.27, so the network has learned the geometric relation. The failure is one of scale in the selection rule, not of perception: the initiation search scores coverage against transition *events*, a handful per fold, and a probability-thresholded gate newly covers fewer of them than a deterministic predicate does, so nine of ten folds reject the candidate for insufficient new coverage.

The Leak-Free Protocol

Re-run on the deduplicated, scene-family-grouped corpus, the meeting fluent yields F1 0.0 on both routes, and mostly by committing a rule that earns nothing rather than by declining to commit one. The direct route selects both_inactive & close on nine of the ten folds, and that clause returns no true positive anywhere (0 tp, 106 fp, 1,812 fn). The Event-Calculus route commits both_active & close on one fold, whose 120 false positives are its entire error mass (0/120/1,812), and abstains on the other nine, recording insufficient new coverage or an abstaining selector; its termination search returns nothing on any fold, most often because no body passed that fold's fit gate. Both routes produce false positives, which an empty theory cannot: this is a transfer failure with the search running, not a search that never ran.The reason is visible in the data. One scene family carries 1,323 of the 1,812 meeting-positive frames and, under honest grouping, sits in exactly one fold; the clause that explains those frames holds *only* there, covering zero of the 489 positives on all eight remaining meeting-bearing segments. Both outcomes follow from that. Where the clause is committed, it is committed on a training side that contains the concentrated family, and it covers nothing outside it. The one fold on which the direct route abstains is precisely the fold that holds that family out, leaving a training side on which the clause covers 0 of 489 positives and 106 negatives. Where the gate abstains, it is because the candidate's held-out score has collapsed for the same reason, not because the statistical bar rose: the per-fold null thresholds under this protocol are essentially those of the like-for-like distributed-corpus run. Training with the family learns a clause that transfers nowhere; training without it cannot learn the clause at all.

Run	P	R	F1
Windowed folds, direct target (mean of 3 folds) precomputed close	—	—	0.7905
learned close_nn	—	—	**0.8276**
Distributed corpus, 10-fold CV, EC + iner pre-tia registered	0.6580	0.8271	**0.7329**
+ termination relations	0.7357	0.8260	0.7782
learned close_nn	0.1250	0.0005	0.0011
Clean protocol (scene-family folds) meeting, EC route	0.0	0.0	0.0
meeting, direct route	0.0	0.0	0.0
moving, EC route	0.0	0.0	0.0
moving, direct route	0.5334	0.3817	0.4450

Table 3: Runs Reported in this Section. Micro F1 Over Pooled Per-Fold Counts, Except the Windowed Rows, Which are Fold Means. The + Termination Relations Row is the Second, Adaptive Vocabulary Iteration. The Cleanprotocol Zeros are Scored Predictions, not Empty Theories, Except the Moving Event-Calculus Row, Whose Initiation Theory is Empty on Every Fold; the Census Behind them is in the Text. Published Figures for the Same Benchmark are Tabulated Separately in Section 10

With eleven observable meeting initiations and five moving ones, the clean corpus does not carry enough evidence for a cross-validated theory in either direction. The machinery is not broken under this protocol. On moving the direct route still selects the canonical published rule shape on nine of ten folds (micro F1 0.4450), and the meeting clause applied to the concentrated family's own fold, an in-family fit and not a transfer test, scores frame F1 0.890 there. What it establishes is narrower and more specific than an abstention: a two-literal state vocabulary does not carry CAVIAR meeting across scene families at all.

Discussion

Two of the findings above concern the method. A perceptual predicate can be learned end-to-end through symbolic credit alone and stand in for hand-set geometry without loss on a real benchmark. And Event-Calculus rules can be induced under fully pre-registered gates and land within a couple of points of the range published Event-Calculus learners report on the corpus they share.The other two concern that corpus. It carries a measurable duplication defect, and under a leak-free split of it the induced meeting rule does not transfer: the search still commits the canonical clause on most folds, and that clause scores zero out of family.None of this ranks the system against the published learners (Section 12). Termination programs are not learned in the pre-registered run either, and the empty termination theory is the dominant source of false positives there. Nor is the learned detector a drop-in replacement for the geometric predicate everywhere: under Event-Calculus cross-validation it is not. And it is one benchmark in one domain. What the section does show is that the structure search and the perception underneath it can be one differentiable object instead of two systems in sequence.

Rule Induction at Scale: Maritime Rendezvous

The previous section ends on an evidence problem: under a leak-free split CAVIAR carries eleven observable meeting initiations, too few for a cross-validated claim about rule learning. This section moves the same machinery to the other public corpus the Event-Calculus learners report on, the Brest AIS maritime stream, whose target fluent rendezVous (two vessels close together, stopped or moving slowly, in open water) carries 3,548 gold intervals.It is large enough to separate three questions CAVIAR could not. Does a crisp rule search, selecting clauses that each fire with weight one as in Section 7, land where the declared ceiling says it will? How much is gained by *weighting* the clauses of the same gated pool instead of selecting them? And what does it cost to learn those weights in one chronological pass over the stream? The six runs of Table 5 answer them, covering the published systems' three training settings: enumerated rule search, weighted clauses, single-pass weight training.Two of the three answers came out better than we expected [24,25].

Corpus and labels

The inputs are two public exports of RTEC , a hand-written Event-Calculus rule engine, over the Brest AIS stream: the composite events it recognised (Zenodo record 2557290) and its own input, the critical points of the compressed trajectories, the timestamps at which a vessel stops, turns, changes speed or loses signal. Those points are the only temporal grid the corpus carries. A deterministic converter reduces the two into a pair-time corpus: one row per (vessel pair, timestamp), eleven relations that hold or do not for that pair then, and the gold label. Table 4 sizes it. The conversion uses no randomness: the corpus is a pure function of the archives. Three properties bound what a number measured on it can mean. The supervision is synthetic: the gold labels are a hand-crafted RTEC rule's output over the same interval streams the vocabulary comes from, so their perfect alignment with those streams says the

conversion Table 4: The pair-time corpus, as the converter leaves it. The negative pairs are drawn by fixed stride from the 2,014 pairs that come within proximity but carry no interval.is faithful, not that the labels are right. The published learners share this regime; the WOLED paper notes that the maritime ground truth is synthetic and its learning curves consequently alike. What every system here measures is the *reconstruction of a known rule* under a vocabulary that does not contain all of it. The vocabulary is incomplete by construction. Its eleven pair-time relations (proximity, both_ lowspeed, both_stopped_far, both_open_sea, became_proximate, and so on) omit the gold rule's two remaining discriminators, its 240s minimumduration threshold and its exclusion of tug and pilot pairs, so perfect reconstruction in the base vocabulary is impossible.And the grid is sparse: ≈1.009 positive rows per gold interval, the covering row almost always the interval's start boundary, so pointwise and interval F1 coincide here (Section 8.4) [26].

Protocol

Metric: The primary metric is pointwise F1 on the positive class, F1 over individual rows, never accuracy, which the empty predictor saturates here. Interval figures accompany it, a gold interval counting as matched when a predicted row falls inside it in the same episode of the pair's timeline, and symmetrically.

The Declared Ceiling: Each run's hypotheses and parameters were fixed before it was executed, beginning with the ceiling the base vocabulary imposes. The gold rule's definitional body is proximity & both_low_or_ stopped & both_open_sea, the middle literal being the disjunction of both_lowspeed and both_stopped_far; all three are in the vocabulary, so the body is expressible either way. It reaches recall 1.0 at pointwise precision ≈0.49, hence F1 ≈0.66: the two inexpressible discriminators are exactly what separate its false positives from its true ones.A direct evaluation of that body on the archives, the ceiling probe, hits the operating point exactly: F1 0.6599, no false negative (Table 6). It also decomposes them: 733 in negative pairs, the pair-exclusion class, 2,956 in predicted runs shorter than 240s, and none outside the two pre-identified discriminators.

Folds are Vessel Pairs

A pair's rows are never split across folds, because the positive mass is concentrated: the top pair carries 33.4% of all positive rows and the top four 60.2%, so a finer split would leak the dominant encounter across the boundary. Five folds, not CAVIAR's ten, are assigned by a deterministic greedy longest-processing-time rule with no seed, stratified by positive mass, none differing from another by more than the largest pair's count. The consequence is disclosed, not smoothed: held-out fold 0 is the top pair plus one negative pair, so per-fold figures are necessarily unequal and the micro number never stands alone.

Search and Gate as On Caviar

The crisp run uses the relational sequential-covering search of Section 7 under the CAVIAR limits: bodies of at most three literals, at most four clauses, per-fold held-out F1 for candidate selection, at least two newly covered positives per clause. Its permutation-null fit gate is the CAVIAR one: 1,000 label permutations per fold, threshold at the 95th percentile of the pool-maximum per-fold F1, derived on the training side. One seed, 7, drives the permutations and the candidate-selection holdout. The target is the direct per-row positive label; induction through inertia is out of scope here, with almost no interior rows to fill.

Weights

The weighted runs keep vocabulary, folds and gate, replacing crisp selection with weights. The body pool is every conjunction of one to three literals passing the same per-fold permutation-null procedure, thresholds re-derived per run, each body carrying one weight. A row's score is the noisy-OR $1 - Q_c$ $1 - \sigma(w_c)\text{cover}_c$ of Section 6.4, every clause purely relational, trained by binary cross-entropy with Adam and thresholded at 0.5: 300 steps at learning rate 0.05 from every weight at −2.0, so every clause starts switched off ($\sigma(-2) \approx 0.12$). The single-pass variants replace that optimisation with *one* pass over the fold in ascending global time, its pairs interleaved as in a stream, in mini-batches of 1,000 rows, one Adam step per batch, at the same rate. Adam is kept instead of WOLED's AdaGrad, so exactly one variable separates the regimes. Pool and gate are still computed on the training side, which makes this single-pass *weight* learning, semi-online in that only the weights see the stream, not online structure learning.

Results

Crisp Search on the Base Vocabulary

The first run is the CAVIAR search unchanged. It lands where the ceiling mechanism predicted, precision near one half at recall near one, for micro pointwise F1 0.6746 (Table 5).On every fold it commits both disjunctive branches of the gold body: both_open_sea & both_stopped_far & proximity and both_lowspeed & both_open_sea & proximity. Three folds add a further clause, on two of them the subsuming disjunction relation itself. No clause is ever accepted below its fold's null gate.The learned operator is not literally the definitional body, its precision being 0.5109 against the body's 0.4924, so the evidence is about the mechanism, not operator identity. The per-fold spread of Table 6 is the pairconcentration effect: the fold with the top pair's long clean encounters scores highest, while folds of many small pairs concentrate the duration-threshold false positives. Improvement past this run has to come from expressiveness, not tuning.

Weighted Clauses on the Same Pool

The second run changes one thing: the pool's clauses are weighted, not selected. We expected it to stay within

[0.66,0.70], the ceiling being a property of the vocabulary. The hypothesis was rejected in the favourable direction. Micro F1 is 0.7398, +0.0652 over the crisp run on the same vocabulary, folds and gate, at markedly higher precision and lower recall; the abstract and Section 1 round that gain to 0.065.The expectation was wrong for an instructive reason: the 0.66 ceiling is the F1 of the definitional body's operating point, crisp reconstruction at recall near one, and weights are not confined to it. Training moved the operating point to high precision at lower recall, which carries more F1.Fold by fold the picture is mixed. The weighted run wins the micro figure and the median but only two of the five folds, and its weakest fold is worse than the crisp run's. That weights over the same clauses beat crisp selection of them is the gain the published line attributes to weighting.

Both Runs Again with the Duration Relation

The third and fourth runs add one relation, sustained_240: a row carries it when proximity∩both_low_or_stopped∩ both_open_sea holds continuously through a stretch of at least 240s containing it. The constant is the gold generator's own, so the addition is language parity with RTEC, not a look at the data.The ceiling probe was re-run on the widened vocabulary before any cross-validated run, against an expectation of ≈0.907 on the assumption that duration removes only the short-run false positives. The ceiling came out higher, at F1 0.9969, again with no false negative. Duration also eliminates at least 711 of the 733 negative-pair false positives, the excluded pairs' bodycondition episodes being almost always shorter than 240s. The bound is a lower one because the 22 false positives the widened probe still makes are not decomposed by class. Two consequences were predicted from that: the pair exclusion is largely redundant with duration here, and the duration vocabulary saturates, so the informative weighted-against-crisp comparison is the base-vocabulary one.Both held. Crisp search on the widened vocabulary lands exactly on the derived ceiling and the weighted run equals it on every fold, so the directional hypothesis, weighted no worse than crisp on the median, holds trivially, with no headroom for weights.

Weighted Clauses Learned in a Single Pass

The last two runs change only how the second run's weights are trained, replacing batch optimisation with a single chronological pass. They use the base vocabulary only: sustained_240 at time t depends on how long the current interval will last, a leak in a streaming setting, and was excluded before the run.Three hypotheses were fixed beforehand: the single pass should lose at most 0.04 micro F1 against the batch run, with anything below 0.70 a negative result; a *reverse* chronological pass should land within 0.02 of the forward one; and the prequential error, scored on each batch before that batch is trained on, should fall along the stream.The measured degradation is zero. The single pass reproduces the batch run's thresholded predictions exactly: the same counts on every fold and, as re-scoring with the stored weights confirms, the same decision on every row, at the same micro F1. The reverse pass is identical as well, and the prequential error falls from the first half of the stream to the second on all five folds in both orders (forward, fold 0: 0.0078 → 0.0036). Halves are the one criterion of these runs settled as the results were written up instead of before them.

Identical predictions are a statement about the decision structure, not evidence that the single-pass path silently fell through to the batch one: the weights themselves differ substantially between the three regimes (Table 5). The mechanism is simpler than robustness to the training regime. In all fifteen fold-by-regime cases exactly one body has $\sigma(w) > 0.5$, always became_proximate & both_open_sea & both_stopped_far, and the secondranked body stays between 0.15 and 0.46, so the noisy-OR of the others never reaches the threshold on any held-out row. All three trainings reduce to the same crisp rule, and the predictions agree because that rule does.The agreement is fragile. The highest score of a row *not* covered by the top body reaches 0.493 in fold 1 under the batch weights, 0.007 below the threshold.That fold has 1,270 rows between 0.3 and 0.5: a small change of learning rate, step count or initialisation could push the second body over and move the counts by hundreds of rows. The declared scope is narrow. Prediction identity holds for this pool, corpus and threshold, hangs on the discrete structure of a thresholded decision, and is not a theorem.

Discussion

The published figure for rendezVous on this corpus is 0.98 for oled and both variants of woled, MLNand ASP-based alike, with theories of eighteen literals, measured on a train/test split of halves. A sec-F1, fold Run PR F1median Base vocabulary (11 relations; declared ceiling ≈0.66)crisp search 0.5109 0.9925 0.6746 0.6596 weighted, batch 0.8715 0.6426 0.7398 0.6928 weighted, forward pass 0.8715 0.6426 **0.7398** 0.6928 weighted, reverse pass 0.8715 0.6426 0.7398 0.6928 Duration vocabulary (+sustained_240; ceiling 0.9969)crisp search 0.9939 1.0 0.9969 0.9950 weighted, batch 0.9939 1.0 0.9969 0.9950

F1, fold				
Run	**P**	**R**	**F1**	**median**
Base vocabulary (11 relations; declared ceiling ≈0.66)				
crisp search 0.5109 0.9925 0.6746 0.6596 weighted, batch 0.8715 0.6426 0.7398 0.6928 weighted, forward pass 0.8715 0.6426 0.7398 0.6928 weighted, reverse pass 0.8715 0.6426 0.7398 0.6928				
Duration vocabulary (+sustained_240; ceiling 0.9969)				
crisp search 0.9939 1.0 0.9969 0.9950 weighted, batch 0.9939 1.0 0.9969 0.9950				

Table 5: Maritime rendezVous: Five-Fold Cross-Validation Over whole Vessel Pairs, Seed 7. P, R and F1 are Pointwise Micro-Averages Over Pooled Per-Fold Counts; the Last Column is the Median of the Five Per-Fold Values. The Three Base-Vocabulary Weighted Rows are Identical Row by Row, Though their Weights are not: Fold 0's Top Body Carries σ(w) = 0.6476 in Batch, 0.5880 in the Forward Pass and 0.8398 in the Reverse, and the L∞ Distance Between the Logit Vectors of Any Two Regimes is 1.06–2.95. Interval-Level F1: 0.6772 (Crisp, Base; 3,545 of the 3,548 Gold Intervals Matched), 0.7435 (Weighted, Base), 0.9970 (Both Duration Rows). No Published Figure Appears Here (Section 8.4)

Ond published experiment, on a six-sequence fragment, compares woled-asp with batch learners and reports figures from 0.91 to 0.97 in a bar chart. None of them stands beside the rows of Table 5, because the two sets of numbers are not on one axis.Theirs live on a dense grid, the critical points of every vessel with inertia filling each interval's interior. Ours live on the sparse per-pair grid the converter produces, and we do not reproduce the published per-timepoint reading on it. A split of halves is not a pair-level five-fold cross-validation either, and for the same reason we do not compare pass times with published training times.What is comparable is comparable in kind, not magnitude. The published systems saturate on their preprocessing of the same stream, and ours too once the duration discriminator is expressible, at the declared ceiling. And the gain the published line reports between OLED and WOLED on CAVIAR, and attributes to weighting, is reproduced here as +0.0652 micro F1 with everything else held fixed, on a corpus with 3,548 gold intervals instead of eleven events, and again without loss when the weights are learned in one pass. Within those bounds the findings are concrete. The crisp search reconstructs the gold body on every fold of a corpus two orders of magnitude richer in positive events, and lands on the declared ceiling. Weighting the same pool adds 0.0652 micro F1 and 0.0332 to the median against the expectation declared for it, on two folds of five. Duration saturates both search modes and makes the pair exclusion largely redundant. And single-pass weights reproduce the batch predictions exactly, for the Reason Named Above.

Base vocabulary	**f0**	**f1**	**f2**	**f3**	**f4**
crisp search	0.9933	0.4690	0.6685	0.5699	0.6596
weighted, all three	0.9346	0.4237	0.8000	0.6928	0.5342
Pooled counts	tp / fp / fn				
ceiling probe, base	3,579 / 3,689 / 0				
crisp search	3,552 / 3,400 / 27				
weighted, all three	2,300 / 339 / 1,279				
ceiling probe, duration	3,579 / 22 / 0				

Table 6: Per-Fold F1 on the Base Vocabulary and the Pooled Counts Behind Table 5. The Three Weighted Regimes Share a Row Because their Predictions are Identical. The Probe Rows Evaluate the Gold Rule's Definitional Body Directly; the Duration Probe's 22 Residual False Positives are what its 0.9969 Ceiling Reflects

Reason Named Above.
No ranking against OLED or WOLED on rendezVous follows from any of it (Section 12). Nor does it say anything about detecting rendezvous in the wild: every number here is a rule-reconstruction figure. It does not demonstrate online structure learning, the pool and gate being computed on the training side, nor a general robustness of thresholded noisy-OR predictors to the training regime. And it exercises none of the GPU substrate:

every run here is a deterministic CPU procedure.What it does show is that the induction protocol of Section 7, its declared gates, pair-level folds and permutation nulls, carries intact to a corpus where its verdicts are measurements and not anecdotes. There, weighted clauses and single-pass training, the two mechanisms the published line rests on, are measured in isolation and behave as their authors report.

Epistemic and Other Reasoning Modes
The substrate generalizes beyond what is true or probable. This section covers three further modes that reuse the

same CUDA join, circuit, and CDCL machinery (epistemic reasoning over world views, probabilistic aggregates, and well-founded semantics) broadening what a neural network can be integrated with while keeping data-plane execution GPU-backed.

Modal Surface and the Epistemic IR

Epistemic logic programming asks what is known or possible across the multiple stable models a program admits— the natural setting for introspection and reasoning under incomplete knowledge. A program may use four modal literals over a predicate: know, possible, not know, not possible. Instead of rewriting them away at parse time, the compiler preserves them in an Epistemic IR (EIR) between the AST and execution planning. The high-level dispatcher classifies modal dependencies before selecting Generate–Propagate–Test, a mode-specific positive-cycle fixpoint, or GPU-backed WFS. The accepted surface spans finite nested modal chains, epistemic integrity constraints (which prune candidate world views on the GPU), rules that couple several epistemic predicates, same-name multi-arity modal predicates, and recursive epistemic execution.

World-View Semantics

A world view is a non-empty set of stable models. know p holds when p appears in every model of the world view; possible p holds when it appears in at least one. Two semantics are selectable. Under the default FAEEL(Founded Autoepistemic Equilibrium Logic), a circular derivation contributes a tuple only when independent founded support exists. A program containing only p :possible p. therefore executes to an empty founded extension instead of failing with an unsupported-construct diagnostic. A Gelfond-1991-style compatibility mode admits that self-supporting tuple. Non-epistemic programs evaluate identically under both.

Epistemic Execution Routes

After EIR construction, acyclic modal programs follow a Generate–Propagate–Test schedule whose phases run as CUDA kernels. *Generate* enumerates bounded candidate assumptions as device bitsets; *propagate* prunes candidates that contradict known or rejected facts; *test* validates survivors against the program's world views, checking stable-model tuple membership on-device per arity. Validating a candidate on this route reuses the on-GPU CDCL solver already built for compilation verification (Section 5), the same substrate that serves satisfiability, bounded MaxSAT and assumption-based queries, and accepted world-view evidence feeds the existing GPUnative exact path.A positive FAEEL dependency cycle reduces to ordinary rules and reaches the founded least fixpoint in the existing recursive engine. A supported Gelfond-1991 cycle, whose selected positive possible gates must use their rule heads' own terms and stay within one recursive dependency component, instead descends to the greatest compatible set: the runtime relaxes those gates for an upper bound and then reevaluates until GPU set comparison over the intensional relations reports no further change (Section 12). Supported cycles through negation take the GPU-backed WFS alternating fixpoint instead, and recursive negation or aggregation inside a compatibility component is rejected, because either would make the descending operator nonmonotone.The reduced ordinary work reuses the optimizer,WCOJ planner and helper-splitting machinery of Section 4, and epistemic splitting decomposes eligible acyclic programs into independent components solved separately and recombined. No CPU fallback is provided for the accepted data-plane hot paths: a construct these routes do not support is rejected with a typed error, not served by a host implementation. Transfer budgets track data movement; the host sequences Gelfond-1991 refinement and WFS iterations, while relation evaluation, snapshot copies and convergence comparisons remain GPU-backed.

Probabilistic Aggregates

Finite aggregate queries (count, sum, min, max, logsumexp) are supported in both exact provenance/PIR inference and Monte Carlo execution, subject to a typed exact-domain cap that rejects domains too large for tractable enumeration. For small finite count domains,aggregate lifting replaces naive world enumeration with exact cardinality dynamic programming over a compact state set (collapsing, for example, a 217-outcome enumeration into a few hundred lifted states with identical results) and accepted count aggregates dispatch a GPUnative exact evaluator rather than a CPU finite-world shortcut, keeping the path device-resident.

Well-Founded Semantics

Programs that violate stratification, such as the classic p :- not q. q :- not p., do not have a unique two-valued least-fixpoint interpretation. For supported shapes, xlog instead assigns three-valued truth (true, false, undefined) through well-founded semantics, computing an alternating fixed point: mark the greatest unfounded set false, derive consequences true, and repeat until stable. For probabilistic programs, true atoms receive normal probability and gradient while false and undefined atoms are assigned probability zero, matching the conservative treatment of non-stratified programs. Probabilistic provenance analysis and epistemic dependency dispatch select the GPU-backed WFS route for the nonmonotone components they support; other cyclic shapes return a typed diagnostic.

Evaluation

This section reports measurements for xlog's deterministic, probabilistic, and neural-symbolic subsystems. The subsystem measurements in Sections 10.2, 10.3 and 10.7 are *single-system* ablations of xlog against its own baselines on development hardware. Section 10.4 then adds controlled head-to-head comparisons against external engines (Scallop, ProbLog2, Soufflé), Section 10.5 tabulates the Event-Calculus rule-induction runs of Section 7 beside the figures

published for the same benchmark and states why the maritime runs of Section 8 are not tabulated beside theirs, and Section 10.6 isolates xlog's two design costs, verification and residency, from ordinary GPU acceleration. Section 10.9 describes three further runtime capabilities for which no timing is claimed.

Methodology

The subsystem ablations were collected on an NVIDIA RTX PRO 3000 (Blackwell, 12GB VRAM, compute capability 12.0). The head-to-head comparisons of Section 10.4 and the overhead isolations of Section 10.6 ran on cloud GPUs, as stated with each, and the neural Event-Calculus runs of Section 10.5 on an NVIDIA A40. The relational searches of Section 10.5 and Section 8 are deterministic CPU procedures whose results do not depend on the host; the weight trainings beside them run on CPU as well. Rust crates are built in release mode; CUDA modules are staged cubin-first with PTX fallback and explicit JIT warm-up; Python components use pyxlog with CUDA-enabled PyTorch. The benchmark harness uses Criterion.rs with the settings in Table 7, and GPU measurements include warm-up to amortize kernel-module loading and CUDA context setup.Per-run measurement data is distributed with the paper for the head-to-head comparisons and overhead isolations, the circuit-cache ablation, the runtimeoptimization ablations of Section 10.7, and every ruleinduction run of Sections 7 and 8. Each other figure below is stated together with the fixture, protocol and aggregation that produced it.

Setting	Value
Sample size	10–100 runs (benchmark-dependent)
Warm-up	3 iterations (PTX JIT, memory-pool init)
Significance level	0.10
Noise threshold	0.05 (ignore <5% variance)
Seeding	Deterministic LCG

Table 7: Benchmark Harness Settings

WCOJ VS. The Binary-Join Plan

This ablation asks how much the worst-case-optimal join path of Section 4 saves over xlog's own binary-join plan on the skewed cyclic query the subsystem exists for. The query is a single three-way triangle join, the classic worstcase-optimal pathology; Table 8 gives the fixture and the timing protocol.The keys are generated by four fixture variants named for the workloads they stand in for (call-graph edges, Andersen points-to , ddisasm-style disassembly , and a neural-symbolic mining analog) whose size parameters coincide at this scale, so the four cells are repeats of one workload and not four distinct ones. What the fixture does establish is skew: every join variable has a small hot domain, so the binary-join plan materializes an intermediate far larger than the final result, which is the regime WCOJ exists to address. WCOJ speedups depend strongly on input size and skew, and synthetic fixtures systematically overstate them, this one included, so the ratio below is the gain on the shape the subsystem targets, at this scale [27,28].

Item	Value
Query	three-way triangle join, one multiway join
Key generation	four fixture variants, coinciding size parameters
Input	three relations of 65,536 rows each
Output	≈4.19M triples
Warm-up	20 windows
Reported time	mean of 10 timing windows
Inner iterations	20 per window (WCOJ), 40 per window (binary join)
Dispersion	coefficient of variation reported, not gated on
Correctness gate	identical row sets on both paths before any timing
Peak allocation	about 2.3GB, within the device's 12GB budget

Table 8: Triangle-Join Fixture and Timing Protocol for the WCOJ Ablation

Figure 4 reports the speedups: a 27.96× geometric mean, with the four cells spanning 26.6× to 29.6×. Since the variants coincide at this scale, that spread measures run-to-run variation and not robustness across distinct workloads. Section 10.4 reports the external counterpart, a fused-counting comparison against Soufflé at millions of tuples.speedup over binary-join baseline (£)

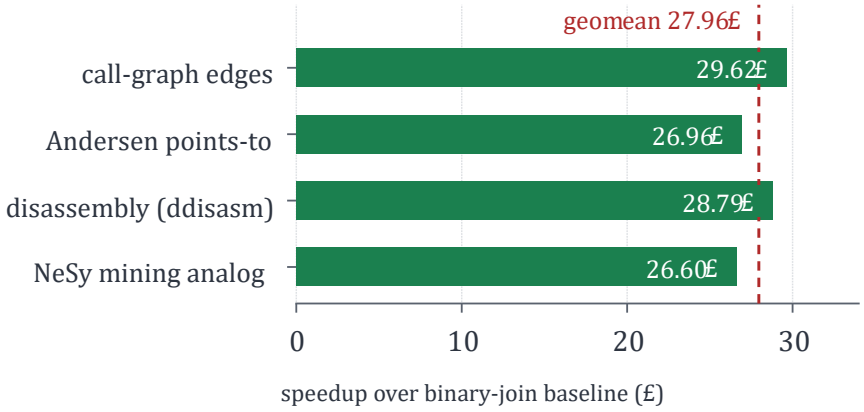


Figure 4: WCOJ VS. Binary-Join Speedup on the Four key-Generation Variants of the Triangle Fixture (Table 8). Single-System Ablation; the Dashed Line is the Geometric Mean

On uniform or empty inputs the adaptive classifier selects the binary-join chain.

Circuit Caching and Training

This ablation asks how much of neural-symbolic training time circuit caching (Section 5) removes by amortizing D4 compilation across epochs. Training is measured on MNIST single-digit addition; Table 9 carries both the full-run timing, in which the first epoch pays coldstart compilation and verification while warm epochs run only the cached forward/ backward kernels, and the cache ablation beside it.The ablation is a separate experiment from the full run: its uncached arm recompiles the circuit every epoch while its cached arm compiles once, so it isolates compilation amortization, and the apparent speedup therefore grows with epoch count toward the cold/warm ratio instead of being a fixed property. The 512-image protocol isolates cold/warm compilation timing and is deliberately under-trained (near-chance accuracy); held-out accuracy is measured at scale in the Scallop head-to-head (Section 10.4), where the same pipeline reaches ≈95%.

Metric	Value	Notes
First epoch (cold)	71.7s	D4 compile + CDCL verify
Steady-state epoch (warm)	0.25s	cached for- ward/backward path
Total training (5 epochs)	72.6s	
Per-query cost (steady state)	0.97ms	forward + backward; steady epoch over 256 pairs, cold epoch excluded
Ablation, uncached arm	242.9s	mean training time, recompiling every epoch
Ablation, cached arm	88.9s	mean training time, compiling once
Circuit-cache speedup	2.74×	95% CI 95% CI [2.29,3.18]

Table 9: MNIST-Addition Training Summary. The Timing Rows Come From a 512-Image, 5-Epoch Training Run at Batch size 64; the Ablation Rows Come From a Separate Experiment (512 Images, 3 Epochs, 3 Seeds), whose Speedup is the Mean of its Three per-Seed Ratios, not the Ratio of its Means

Head-to-Head Comparisons

Table 10 reports-controlled comparisons against external engines, one per reasoning mode, on cloud GPUs (RTX 3090/4090), with matched programs, data and metrics. xlog wins where its GPU-resident, bounded-memory design is the point, and does not where a mature CPU compiler is already fast enough on small inputs.

Neural-Symbolic: MNIST Addition VS. Scallop

With identical MNISTNet, data, metric and seeds, xlog and Scallop reach comparable held-out accuracy, and xlog runs a steady epoch 2.80× faster and finishes ahead end to end once the one-time D4 compilation amortizes. The two systems reach that accuracy by different routes: Scallop scores each fact through a bounded set of proof clauses, while xlog evaluates the exact weighted model count of the compiled circuit on every step. The advantage is time and residency, not accuracy.

Mode / task	Baseline	xlog	Baseline	Ratio
Neural: MNIST addition; 20000 training images, 5 epochs, 3 seeds, held-out on the 10k test set	Scallop	acc 0.955 ± 0.003; steady epoch 8.71±0.09s; 108.5s over 5 epochs	acc 0.950 ± 0.006; steady epoch 24.35 ± 0.51s; 122.7s over 5 epochs	**2.80×** faster per epoch
Probabilistic: exact inference on five small programs	ProbLog2	0 error; per query flat at ≈0.25s	0 error; per query 0.014–0.028s	9–19× slower
Deterministic: Soufflé (standalone compiled mode, 8 jobs) triangle counting on hub-skewed graphs of 3.65M, 7.51M and 9.39M	triangles	counts match at every size; 0.43, 0.37 and 0.45s; peak device allocations 85, 204 and 359MB	1.02, 2.94 and 4.05s; one-time per-case compilation recorded separately	Soufflé/xlog: 2.37×, 7.95×, 9.00×

Table 10: Head-to-Head Comparisons: One External Engine Per Reasoning Mode, Matched Programs, Data, Metrics and Seeds, on Cloud GPU Pods. The Correctness and All-Arm Completeness Gates Pass for Every Comparison

Probabilistic: Exact Inference VS. ProbLog2
On five programs (a conditioned Bayesian fragment and probabilistic reach chains) xlog's exact query probabilities match the analytic answers to zero error, identical to ProbLog2 (correctness gate $|\Delta| < 10{-}4$). On timing, however, xlog is *not* competitive on these small programs. Its per-query time is flat, a fixed GPU launch plus a per-call D4 compile and CDCL verify, while ProbLog2's mature CPU compiler runs each query 9–19× faster. We therefore make no exact-inference *speed* claim; the probabilistic contribution is correctness-equivalent inference with a CUDA-resident circuit data plane and certification of the final smoothed circuit. The host orchestrates the route and observes bounded solver control state, and its fixed costs amortize only at scale or under repeated evaluation.

Deterministic: WCOJ Triangle Counting VS. Soufflé
On hub-skewed graphs, xlog's fused worst-caseoptimal triangle counting, xlog's enumerate-then-count diagnostic, and Soufflé produce the same per-root triangle totals at every size. The comparison uses the median of three full-process executions after one-time native builds: a standalone Soufflé executable is generated once per case with eight jobs, and its 12.18–13.10s compilation is recorded separately. Relative to fused xlog execution, compiled Soufflé is 2.37×, 7.95× and 9.00× slower on the three measured cases. Fused counting never materializes the triangles and its peak device allocations are 85, 204 and 359MB, while enumerate-then-count peaks at 3,287, 8,403 and 15,247MB. All arms complete, both crosssystem count checks pass, and the all-arm comparison gate is therefore true. This supports bounded intermediates on these skewed, large-intermediate workloads, not a universal Datalog speed claim: native compilation time is excluded from the execution medians, the measurements are hardware-specific, and on moderate skew the advantage over xlog's own binary join is only 1.1–1.5×.

Event-Calculus Rule Induction
This subsection asks how the induced theories score against the figures other authors have published for the same benchmark, and what the runs that do not favour the system look like. Section 7 states the induction protocol and analyses its runs, and Table 3 there lists them; Table 12 collects their frame-level scores.

Published Figures for the Same Benchmark
Table 11 places our pre-registered ten-fold row beside the four published frame-level holdsAt F1 figures for meeting on the whole distributed CAVIAR corpus. Our figure is below the floor of the published span, by 0.0021: a fully pre-registered run lands just under the lowest published figure, which is the hand-crafted rule set. The second-iteration run reaches 0.7782, which is inside the span, but it is one disclosed adaptive iteration (Section 7); it is not the row in this table.

The Direct-Protocol Reference on the Same Folds
The run that produced that row also scores a direct pertimestep reference theory on the same ten folds, under the direct protocol's own pre-registered defaults: held-out accuracy, a fixed 0.75 fit gate, the default tie floor. It lands far below the Event-Calculus row beside it (Table 12). The same reference theory comes out of all three ten-fold runs (the pre-registered one, the second iteration and the neural negative), so the second iteration's added vocabulary never reaches the direct search. Its recall is dominated by one fold on which the direct search abstains outright, leaving 855 gold frames undetected. The figure is what those defaults produce on this corpus, not a ceiling for the direct protocol,

whose studied settings produce the windowed-fold results of Section 7.The remaining runs of Section 7, including the two that do not favour the system, are scored in Table 12 and diagnosed there. One of those rows needs a disclosure that section does not carry.

The Moving Threshold, Disclosed

The two fluents carry different canonical proximity thresholds in the published CAVIAR rule set (meeting over close_25, moving over close_34), and the moving figure above is measured under the canonical close_34. An earlier run over the same folds measured moving under the meeting threshold of 25 and scored higher (Table 12). The canonical threshold remains the right headline, and the higher number is not evidence against it: where the clause is selected, widening close from 25 to 34 behaves exactly as threshold semantics predict, recall up and precision down.The micro figure falls for a selection-level reason instead. On the fold carrying 1,546 of the 3,136 gold moving frames, the wider candidate pool puts the top two candidates inside the selection tie band, at a margin of 0.0093 against the 0.01 tie floor, and the search abstains there where the threshold-25 run had committed. That figure therefore leaned on a fold whose selection was tie-fragile, which is a reason to report the lower canonical number and the disagreement together rather than either alone.

The Maritime Figures are not Tabulated Beside the Published Ones

The maritime runs of Section 8 are not placed beside the published figure for that fluent; Section 8.4 says why.

Verification Cost and Residency Benefit

These two isolations ask what xlog's design costs, separated from ordinary GPU acceleration: what the equivalence proof buys with cold-compile time, and what device

System	P	R	F1	Source
Hand-crafted rules	0.644	0.855	0.735	OLED [22] Tab. 1(b), EC_crisp; the same value appears in WOLED [23] Tab. 2
OLED	0.678	0.953	0.792	OLED [22] Tab. 1(b), whole dataset
OLED, as re-run by other authors	—	—	0.782	WOLED [23] Tab. 2
This work, EC + inertia, pre-registered	0.6580	0.8271	0.7329	10-fold CV by video segment, all gates re-derived per fold (Section 7)
WOLED-ASP	—	—	0.887	WOLED [23] Tab. 2

Table 11: Frame-Level HoldsAt F1 for the Meeting Fluent on the Distributed CAVIAR Corpus, Micro-Averaged Over Pooled Counts. Rows are Ordered Alphabetically by System Name, Not by F1: the Five Protocol Deltas Printed Beneath the Table do not all Act in the Same Direction. Dashes are Figures the Cited Table Does not Report

Protocol Deltas, Which Must Travel with this Table

- The published runs cross-validate over their own windowed interpretations with subsampled negatives; ours folds over whole video segments.
- They learn full initiation and termination programs; our row commits a two-literal initiation clause and an empty termination theory. Out of its 2,304 predicted-positive frames, 781 of the 788 false positives fall on the folds with detected interior terminations; the other 7 are initiation-side errors persisted by inertia on a fold whose test side detects none.
- Their rule language admits longer bodies—bottom clauses averaging some fifteen literals against our three-literal cap.
- Their settings are reported as the best among several tried; ours were pre-registered. No number of ours is adjusted on that basis.
- All rows compute F1 on holdsAt inferred through inertia, on the same corpus, the one axis on which the protocols do match, and the reason the rows are placed side by side at all.

Run	Corpus	P	R	F1	tp/fp/fn
meeting, Event-Calculus route, pre-registered	distributed	0.6580	0.8271	0.7329	1516/788/317
meeting, direct-protocol reference, same folds	distributed	0.6864	0.1266	0.2137	232/106/1,601
meeting, Event-Calculus route with learned close_nn	distributed	0.1250	0.0005	0.0011	1/7/1,832
meeting, Event-Calculus route	clean	—	—	0.0	0/120/1812

meeting, direct route	clean	—	—	0.0	0/106/1812
moving, Event-Calculus route, no initiation clause on any fold	clean	—	—	—	0/0/3136
moving, direct route, canonical close_34	clean	0.5334	0.3817	0.4450	1197/1047/1939
moving, direct route, earlier run under close_25	clean	—	—	0.4868	1295/890/1841

Table 12: Frame-Level HoldsAt Scores for Every Rule-Induction Run of Section 7, Micro F1 Over Pooled Per-Fold Counts. The Distributed Corpus is the Dump Shipped with the Published Event-Calculus Learners and Carries the Duplication Defect Measured in Section 7; the Clean Corpus is its Deduplicated, Scene-Family-Grouped Form. Dashes are Quantities the Run Does Not Report Separately

residency saves in transfers. Both run on an RTX 3090, and Table 13 carries their measurements.

Verification Cost: The cold epoch's compilation time is dominated by the on-GPU CDCL equivalence check, not the D4 compile. Split on probabilistic reachability chains, CDCL verification accounts for 96–100% of the cold-compile time and grows with circuit size, while D4 compilation stays under 17ms. The "expensive" cold epoch is thus almost entirely the price of the machinechecked correctness certificate (Section 5), a cost the CPU-symbolic and GPU-Datalog systems surveyed in Section 11 do not pay, because they do not verify.

Setting	Measured
Verification cost chain n=5	CDCL verify 251ms
chain n=40	CDCL verify 508ms
Residency benefit per handoff	round-trip of roughly 40–56µs
32 handoffs/step	4% of the step
128 handoffs/step	≈10% of the step; this is the standard batch-64 MNIST-addition step, 7.2ms of 72ms
512 handoffs/step	holds near 9% of the step

Table 13: Design-Cost Isolations. Verification Splits the Cold Compile on Probabilistic Reachability Chains of Length N. Residency Forces a Device→Host→Device Round-Trip at the Neural–Symbolic Boundary, Swept Over 2 to 512 Handoffs Per Step on the Batched Addition Path

Residency Benefit

To isolate the transfer-free advantage from ordinary GPU acceleration we run the same neural–symbolic pipeline with and without a forced device→host→device round-trip at the neural–symbolic boundary, the transfer a CPU-reasoning hybrid pays every step, sweeping the number of neural→symbolic handoffs per step. Below a handful of handoffs the roundtrip cost is lost in run-to-run variation. From a few dozen upward it is a visible share of the step, because the batched circuit compute grows sublinearly with the batch while the transfer grows linearly with the handoff count.On a single query the transfer is therefore negligible; at a realistic training batch it is a ≈10% tax that xlog does not pay, distinct from and additive to the GPU-compute and circuit-caching advantages. The forced per-buffer copy is an upper bound, since a hybrid could coalesce transfers. The larger transfer-dominated regime, largestate Datalog fixpoints where whole relations would cross the bus every iteration (Section 1), lies outside what this isolation measures (Section 12).

Index Reuse and the Chain Scorer

This pair of ablations asks what two of the runtime optimizations of Section 4 save on the repeated-session workloads they target, each against the path it replaces. The persistent hash-index manager retains built join indices across evaluations, keyed on relation generation, schema signature and device ordinal, under a byte-budgeted LRU policy: a session whose relations change little between calls reuses an index instead of rebuilding it, and a generation change invalidates the entry instead of serving a stale one. The profile-gated shared-memory chain scorer tiles the left relation through shared memory when scoring chain-shaped candidate bodies during exact rule induction, and is gated on a candidate-row threshold below which the baseline scorer runs unchanged. Both target the same waste, work repeated across calls that the previous call already did, and both are transparent to the program. Common-subexpression elimination and adaptive re-optimization are held to output equivalence with the unoptimized plan.Table 14 gives the medians. The index fixture is buildheavy by construction, so that building the index dominates the join, and neither arm records a host transfer. The chain scorer's two arms do not overlap on the chainhot fixture; on a small input below its gate threshold, where the baseline path runs instead, the two medians differ by 1.3% and the arms' ranges overlap. Both are point medians on single synthetic fixtures, and the scorer's coverage output is identical to the baseline's on both.

Fixture	Baseline median → optimized median
Persistent index reuse: repeated-session semi-join, eight left rows probing eight million right rows	0.255s → 0.0794s, a 3.21× gain over 9 timed runs after 12 warm-up runs
Shared-memory chain scorer: chain-hot body search, 768 candidate rows against a gate threshold of 256	27.51ms (27.3–28.3) → 4.93ms (4.9–5.4), a 5.58× gain over 12 timed runs after 3 warm-up runs

Table 14: Runtime-Optimization Ablations, Each a Median Over the Timed Runs of One Synthetic Fixture

Capability Coverage Across Systems

Table 15 positions xlog qualitatively against representative systems; each addresses a subset of the design space, and xlog's contribution is exposing the supported routes through one typed language and provider-owned CUDA runtime services. This table is capability coverage, not performance: the quantitative head-to-head results against Scallop, ProbLog2 and Soufflé are in Section 10.4.The rule-induction row covers two paths of different maturity: the trainable-rule-body path is carried onto an external benchmark in Section 7 and tabulated in Section 10.5, while the differentiable-ILP path is not exercised there at all (Section 6). The table's "yes" claims capability, not maturity (Section 12).

Dimension	DeepProbLog	Scallop	Lobster	GPUlog	xlog
Symbolic execution	CPU (Prolog)	CPU (Datalog)	GPU (Datalog)	GPU (Datalog)	GPU (semi-naive)
Probabilistic inference	Exact (CPU)	Provenance (CPU)	Provenance (GPU)	None	Exact d-DNNF + (GPU)
Circuit verification	None	None	None	—	On-GPU CDCL
Neural integration	Prolog bridge	Differentiable	Differentiable (GPU)	None	Zero-copy, fused
Zero-copy ML interop	No	Partial	Partial	No	Yes
Rule structure induction	No	No	No	No	Yes (beta): trainable bodies, dILP
Epistemic reasoning	No	No	No	No	Yes (finite fragment)

Table 15: Capability Comparison Across Neurosymbolic and GPU Logic Systems

Log-Evidence, Rule unions, and Exact Constraint Solving

The capabilities below round out the runtime surface. Each is described for what it does; no timing is claimed for any of them.

Exact Log-Evidence and the CNF-Variable-To Fact Map: An exact evaluation now returns the log partition function of the weighted formula, log$Z$, alongside the per-query probabilities, so a program's evidence mass is read off directly instead of being reconstructed from marginals; it surfaces as log_z_e on the Python result. Paired with it, prob_var_map() reports which probabilistic fact each CNF variable stands for (an ordinary fact, one Bernoulli decision of an annotated disjunction's chain, or padding) so the per-variable gradient buffers can be attributed back to program facts and not to opaque indices. Its length is the encoder's variable *capacity*, not a variable count, and it is refused with a typed error where there is no CNF encoding to report: Monte Carlo programs, and exact programs taking the GPU fast path that skips the encoding.

Same-Head Rule Unions in One Multiway Pass: A predicate head defined by many rules used to fold its per-rule contributions into the head relation one union at a time, re-sorting the growing accumulator once per rule and going quadratic in the rule count, a real cost at corpus scale where one head can carry thousands of rules. Contributions are now merged with a single multiway union per head per iteration. The union count therefore no longer scales with the same-head rule count (eight same-head rules record one union, not eight) while the derived rows stay byte-identical to the unbatched result.

An Exact Stage Beyond Enumeration Capacity: Constraint solving over several interacting constraints enumerates a component completely where it is small enough. Long chain-shaped components that exceed that capacity take a memoized dynamic-programming stage on the GPU instead, which computes the same totals exactly instead of approximating them. A component wider than the configured limit is refused with a typed error instead of served by an approximation, and the solver's buffers reach Python through the same zero-copy DLPack views as the rest of the runtime (Section 6.3).

Related Work
xlog draws on five lines of research: neurosymbolic programming, GPU-accelerated Datalog, probabilistic logic programming, GPU SAT, and differentiable ILP. Its contribution is to expose those capabilities through one typed language and shared CUDA services while retaining routespecific execution and differentiability contracts.

Neurosymbolic Programming
DeepProbLog replaces selected probabilistic facts with neural-network outputs, enabling end-to-end gradient training of neural predicates inside a probabilistic-logic framework. NeurASP integrates network outputs as probability annotations on answer-set programs. Both perform symbolic inference on the CPU and shuttle gradients across the bus. Scallop provides a differentiable Datalog with provenance-semiring reasoning and a relational symbolic representation, and is the closest language-level analogue to xlog. Its reasoning core, however, is CPU-based.Lobster is the most directly comparable system: it GPU-accelerates neurosymbolic programs in the Scallop style, and shares xlog's premise that the symbolic data plane belongs on the GPU. Against it, xlog is complementary along three axes. It includes exact knowledge compilation and epistemic reasoning in addition to provenance-based differentiable Datalog. It certifies the final smoothed circuit's logical equivalence to its backend source formula using GPU CDCL and on-device proof checking. And it reports route-specific residency with byte-level transfer accounting. No head-to-head performance comparison with Lobster is drawn here (Section 12).Differentiable-SAT and logic-as-loss methods (SAT-Net, semantic loss , and Logic Tensor Networks instead couple neural models to constraints through relaxed or fuzzy surrogates. By contrast, xlog couples through exact weighted model counting, trading their broader applicability for exact gradients and the on-device verifiability of Section 5 [29-33].

GPU-Accelerated Datalog and Joins
GPUlog ported bottom-up fixed-point computation to CUDA using HISA indexing, and reported up to 45× speedups over CPU Datalog engines. VFLog advanced this with a vertically-fused, column-oriented kernel design that avoids intermediate materialization, reporting up to 200× over CPU column engines; mnmgDatalog extends GPU Datalog to multi-node, multi-GPU clusters. Most relevant to xlog's join path, Sun et al. scale worst-case-optimal joins to GPUs. All of them target deterministic Datalog only: none supports probabilistic semantics, weighted model counting, or gradient computation over derived facts. Like them, xlog uses a columnar, kernel-fused execution model, and it additionally promotes eligible cyclic and multiway rules to a worst-case-optimal join path. But that engine is one component of a substrate that extends through circuit-based inference to neural-symbolic gradient propagation in a single address space [34,35].

Probabilistic Logic Programming
ProbLog2 established the modern pipeline for exact probabilistic inference: ground the program, encode provenance as a propositional formula, compile to a tractable target (d-DNNF or SDD , and evaluate weighted model counts . The compilers (D4 , c2d run as host-side subprocesses, and recent work makes such compilation certified . xlog adopts the same compile-once/evaluate-many model with CUDAbacked data planes for provenance, Tseitin encoding, D4 compilation, equivalence verification, and forward/backward evaluation. The host orchestrates those stages and reads bounded solver control state; device circuit and gradient buffers share an address space with the neural and deterministic data planes [36,37].

GPU SAT
ParaFROST parallelizes CDCL clause-database simplification on the GPU while retaining sequential conflictdriven search on the CPU. FastFourierSAT reformulates SAT as continuous optimization and applies GPU local search, an incomplete solver for satisfiable instances.Both are standalone competition solvers. In xlog the GPU CDCL module instead serves as a verification component inside knowledge compilation (Section 5), proving circuit–formula equivalence via two UNSAT proofs and providing the certificates the probabilistic and epistemic backends rely on [38].

Differentiable ILP
Differentiable ILP casts first-order rule induction as differentiable optimization, scoring candidate rules by soft forward-chaining. Dense materialization over the rule space scales cubically in the number of constants, and implementations remain CPU-based. The dILP subsystem of xlog (Section 6) replaces dense materialization with sparse GPU bitmasks and on-device scatter–gather credit assignment, avoiding the cubic blowup while preserving differentiability [39].

Logic Programming Languages
xlog inherits ideas from a long line of logic languages. Prolog offers unification and metaprogramming on a CPU interpreter; Mercury introduced strong typing and modes, compiling to native code. Soufflé is a typed Datalog compiling to C++ for program analysis, and clingo provides answer-set semantics on CPU grounders and solvers. From these xlog adopts the typed-predicate discipline while unifying typed Datalog, probabilistic facts, neural predicates, and epistemic operators in one language whose supported routes use CUDA kernels and provider-owned runtime services [40-43].

Limitations and Future Work
Current Limitations
NVIDIA GPU Required: The execution kernels are written in CUDA C; there is no OpenCL, Metal, or ROCm backend.

The persistent-index manager, recordedlaunch discipline, and in-kernel hash-consing use CUDAspecific primitives such as cooperative groups, warp-level operations, and unified virtual addressing. Users without a supported NVIDIA GPU cannot run xlog.

Device Memory and Fixed Capacities are Bounded: Relations, indices, circuit buffers, and Monte Carlo sample arrays are allocated on-device; there is no out-of-core spilling. An allocation that exceeds the configured byte budget fails with resource_exhausted. A row, tuple, sample, domain, or other fixed-count limit instead fails with capacity_ exceeded, naming the exhausted unit. A bounded recursive or Monte Carlo fixpoint that reaches its iteration limit without convergence fails with convergence failure; partial counters do not turn it into a result.

Maturity of the Rule-Learning Surfaces: The differentiable-ILP subsystem (Section 6) is beta: the search space is restricted to definite programs, convergence is sensitive to learning-rate schedules, and its API may change. The trainable-rule-body surface of Section 6.4 (mixed neural/symbolic bodies, existential joins, and joint multi-rule mixtures) is at the same maturity. It is carried onto a public benchmark under pre-registered gates and cross-validation in Section 7, but how its accuracy and training cost compare with dedicated relational learners on standard benchmarks is an open question.

Coverage of the Epistemic Runtime: The epistemic runtime (Section 9) executes its accepted dataplane work on the GPU, but the broader solver portfolio (MaxSAT/portfolio services) and the end-to-end probabilistic–epistemic evidence path are still being broadened. The negated-modal WFS route and Gelfond1991 compatibility refinement are GPU-backed but hostorchestrated. Positive modal dependency cycles have defined execution: FAEEL computes a founded least fixpoint, including an empty extension for an unseeded cycle, and supported Gelfond-1991 positive possible cycles compute the greatest compatible set of concrete tuples from an upper bound and frozen snapshots. The shared recursion-depth setting bounds both descending compatibility refinements and WFS iterations.

Fail-Closed Boundaries: Raw lowering of modal literals, recursive epistemic programs that also contain modal integrity constraints, WFS shapes outside the supported negated-modal plan, unbounded tuple-key forms, and recursive negation or aggregation within a Gelfond1991 compatibility component all remain fail-closed. The Python batch-query helpers are typed but do not yet accept floating-point uploads, a convenience-API gap and not a core execution limit.

Partial Head-to-Head Coverage: Section 10.4 reports controlled comparisons on identical programs and data against one external engine per reasoning mode: Scallop (neural-symbolic), ProbLog2 (probabilistic), and Soufflé (deterministic). Coverage is still partial. Lobster, the most directly comparable GPU-neurosymbolic system, and the GPU Datalog engines GPUlog and VFLog are not among the baselines; DeepProbLog was not run under a matched (non-timeout) protocol; and each comparison rests on a single GPU configuration, not a crosshardware sweep. Broader multi-system, multi-hardware comparison remains future work.

Rule-Induction Evidence: The evidence of Section 7 is thinnest exactly where it is leak-free: the deduplicated, scene-family-grouped corpus carries too few initiation events for a cross-validated claim. Both routes commit a clause on most folds and that clause does not transfer, so the protocol yields no meeting theory and supports no conclusion about that fluent's learnability in either direction. The protocol-matched numbers are computed instead on the dump distributed with the published Event-Calculus learners, which carries a measured duplication defect; we report there because it is the regime the published figures share, and every figure computed on it inherits that caveat. No ordering against OLED, WOLED-ASP or the hand-crafted rule set follows from Table 11, and none is claimed: the five protocol deltas printed with that table are real, unquantified, and do not act in the same direction, while the clean protocol has too few events to settle any of them. Learning termination programs and lifting the body-length cap, which would remove two of those deltas, is future work.

Two Figures of Different Status: The headline figure of Section 7 is fully pre-registered: gates, seeds and folds were fixed before the run, and that run was executed once. The second-iteration figure differs from it in exactly one respect, its Event-Calculus vocabulary having been chosen after the first run's failure mode was known on the same folds, which makes it one disclosed adaptive iteration and not a pre-registered result; it is reported only with that label and is never the row we place beside a published number.

Maritime Scope: The maritime runs of Section 8 remove the evidence-starvation bound, but they carry bounds of their own. Their supervision is synthetic: the gold labels are a hand-crafted rule's output over the same streams the vocabulary is derived from, a regime the published maritime figures share. Every number there is therefore a rule-reconstruction figure under an incomplete vocabulary, not a detection figure, and none of those runs exercises the GPU substrate. Their temporal grid is the critical-point grid of the RTEC exports, not the dense grid of the published figure, so no maritime number of ours stands beside the published 0.98 and no pass time beside the published training times (Section 8.4). And the single-pass runs are online in the weights only, their body pool and permutation-null gate being computed on the training side: they demonstrate single-pass weight learning and not online structure revision, which remains future work, and the zero degradation they measure rests on a single body sitting above the decision threshold in every regime, on a 0.007 margin; it is a property of this pool, corpus and threshold, not a robustness result.

Future Work

Out-Of-Core Execution: A streaming mode would partition relations across host and device, paging tiles through GPU memory in fixpoint-iteration order, removing the single-GPU-memory ceiling at the cost of added transfers.

Multi-GPU Partitioned Evaluation: Inspired by mnmgDatalog's radix-hash partitioning and GPUaware all-to-all shuffle, a multi-GPU backend would distribute relations across devices and coordinate joins over NVLink or PCIe.

Deeper Epistemic Residency: The remaining epistemic work is stronger residency and coverage: a deviceresident, no-host-interaction well-founded path, broader non-finite tuple-key forms, and semantic forms beyond the current finite accepted surface.

Conclusion

xlog is a CUDA-native logic programming engine with one typed language and provider-owned runtime services for deterministic, probabilistic, epistemic, and neuralsymbolic use. A neural network is an ordinary predicate; exact query gradients flow through the probabilistic knowledge-compilation route, while the other routes reuse typed device-buffer interfaces without claiming identical differentiability. The final smoothed probabilistic circuit is certified against its back-end source formula before it is cached or evaluated. Solver proof traces are checked on the GPU, and the host observes bounded status and error scalars.

Residency claims are likewise route-specific. Ordinary relational and exact execution are host-orchestrated over device data planes. Eligible recursion and resident Monte Carlo use certified cores that record zero tracked transfers and then return one authoritative terminal receipt. Byte exhaustion, fixed-capacity exhaustion, and nonconvergence remain separate typed outcomes.

The measurements describe a system that pays off where the symbolic workload is large and repeated: reusing a verified circuit across training iterations, and avoiding the intermediate blowup of binary-join plans on skewed cyclic queries. Against external engines the picture is mixed: clear gains in bounded-memory join evaluation and in per-epoch neurosymbolic training, and a clear loss to a mature CPU compiler on small exactinference programs. What they do not give is a general ranking of xlog against other systems. Each comparison covers one engine per reasoning mode on one class of program, the rule-induction results rest on two public corpora, and the epistemic and rule-learning surfaces are young. The evidence supports the architecture, not a claim of universal superiority.The direction is the one the architecture already implies: out-of-core and multi-GPU execution to lift the memory ceiling, deeper residency for the epistemic routes, and structure learning that runs on the stream rather than over it.

References

1. Manhaeve, R., Dumancic, S., Kimmig, A., Demeester, T., & De Raedt, L. (2018). Deepproblog: Neural probabilistic logic programming. Advances in neural information processing systems, 31.
2. Zhun Yang, Adam Ishay, and Joohyung Lee. "NeurASP: Embracing Neural Networks into Answer Set Programming". In: Proceedings of the Twenty-Ninth International Joint Conference on Artificial Intelligence (IJCAI). 2020.
3. Sun, Y., Shovon, A. R., Gilray, T., Kumar, S., & Micinski, K. (2025, March). Optimizing Datalog for the GPU. In Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1 (pp. 762-776).
4. Sun, Y., Kumar, S., Gilray, T., & Micinski, K. (2025, April). Column-oriented datalog on the GPU. In Proceedings of the AAAI Conference on Artificial Intelligence (Vol. 39, No. 14, pp. 15177-15185).
5. Osama, M., Wijs, A., & Biere, A. (2021, March). SAT solving with GPU accelerated inprocessing. In International Conference on Tools and Algorithms for the Construction and Analysis of Systems (pp. 133-151). Cham: Springer International Publishing.
6. DLPack: Open In Memory Tensor Structure. Accessed 2026.
7. cudarc: Safe Rust Wrapper around CUDA. Accessed 2026.
8. PyO3: Rust Bindings for Python. Accessed 2026.
9. Ngo, H. Q., Porat, E., Ré, C., & Rudra, A. (2018). Worst-case optimal join algorithms. Journal of the ACM (JACM), 65(3), 1-40.
10. Veldhuizen, T. L. (2014, March). Leapfrog triejoin: A simple, worst-case optimal join algorithm. In Proc. International Conference on Database Theory.
11. Wang, Y. R., Willsey, M., & Suciu, D. (2023). Free join: Unifying worst-case optimal and traditional joins. Proceedings of the ACM on Management of Data, 1(2), 1-23.
12. Fierens, D., Van den Broeck, G., Renkens, J., Shterionov, D., Gutmann, B., Thon, I., ... & De Raedt, L. (2015). Inference and learning in probabilistic logic programs using weighted boolean formulas. Theory and Practice of Logic Programming, 15(3), 358-401.
13. Green, T. J., Karvounarakis, G., & Tannen, V. (2007, June). Provenance semirings. In Proceedings of the twenty-sixth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems (pp. 31-40).
14. Grigori S. Tseitin. "On the Complexity of Derivation in Propositional Calculus". In: Studies in Constructive Mathematics and Mathematical Logic (1968).
15. Darwiche, A. (2001). Decomposable negation normal form. Journal of the ACM (JACM), 48(4), 608-647.

16. Lagniez, J. M., & Marquis, P. (2017, August). An Improved Decision-DNNF Compiler. In IJCAI (Vol. 17, pp. 667-673).
17. Moskewicz, M. W., Madigan, C. F., Zhao, Y., Zhang, L., & Malik, S. (2001, June). Chaff: Engineering an efficient SAT solver. In Proceedings of the 38th annual design automation conference (pp. 530-535).
18. Darwiche, A., & Marquis, P. (2002). A knowledge compilation map. Journal of Artificial Intelligence Research, 17, 229-264.
19. Chavira, M., & Darwiche, A. (2008). On probabilistic inference by weighted model counting. Artificial Intelligence, 172(6-7), 772-799.
20. Capelli, F. (2019, June). Knowledge compilation languages as proof systems. In International Conference on Theory and Applications of Satisfiability Testing (pp. 90-99). Cham: Springer International Publishing.
21. Apache Arrow: Cross-Language Development Platform for In-Memory Data Accessed 2026.
22. Katzouris, N., Artikis, A., & Paliouras, G. (2016). Online learning of event definitions. Theory and Practice of Logic Programming, 16(5-6), 817-833.
23. Katzouris, N., Paliouras, G., & Artikis, A. (2023). Online learning probabilistic event calculus theories in answer set programming. Theory and Practice of Logic Programming, 23(2), 362-386. arXiv: 2104.00158.
24. Ray, C., Dréo, R., Camossi, E., Jousselme, A. L., & Iphar, C. (2019). Heterogeneous integrated dataset for maritime intelligence, surveillance, and reconnaissance. Data in brief, 25, 104141.
25. Pitsikalis, M., Artikis, A., Dreo, R., Ray, C., Camossi, E., & Jousselme, A. L. (2019, June). Composite event recognition for maritime monitoring. In Proceedings of the 13th ACM international conference on distributed and event-based systems (pp. 163-174).
26. Alexander Artikis, Marek Sergot, and Georgios Paliouras. "An Event Calculus for Event Recognition". In: IEEE Transactions on Knowledge and Data Engineering 27.4 (2015), pp. 895–908.
27. Andersen, L. O. (1994). Program analysis and specialization for the C programming language. PhD Thesis, University of Copenhagen.
28. Flores-Montoya, A., & Schulte, E. (2020). Datalog disassembly. In 29th USENIX Security Symposium (USENIX Security 20) (pp. 1075-1092).
29. Li, Z., Huang, J., & Naik, M. (2023). Scallop: A language for neurosymbolic programming. Proceedings of the ACM on Programming Languages, 7(PLDI), 1463-1487.
30. Biberstein, P., Li, Z., Devietti, J., & Naik, M. (2026, March). Lobster: A gpu-accelerated framework for neurosymbolic programming. In Proceedings of the 31st ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1 (pp. 147-162).
31. Wang, P. W., Donti, P., Wilder, B., & Kolter, Z. (2019, May). Satnet: Bridging deep learning and logical reasoning using a differentiable satisfiability solver. In International Conference on Machine Learning (pp. 6545-6554). PMLR.
32. Xu, J., Zhang, Z., Friedman, T., Liang, Y., & Broeck, G. (2018, July). A semantic loss function for deep learning with symbolic knowledge. In International conference on machine learning (pp. 5502-5511). PMLR.
33. Badreddine, S., Garcez, A. D. A., Serafini, L., & Spranger, M. (2022). Logic tensor networks. Artificial Intelligence, 303, 103649.
34. Shovon, A. R., Sun, Y., Micinski, K., Gilray, T., & Kumar, S. (2025, June). Multi-node multi-gpu datalog. In Proceedings of the 39th ACM International Conference on Supercomputing (pp. 822-836).
35. Sun, Y., Qi, K., Gilray, T., Kumar, S., & Micinski, K. (2026). Scaling Worst-Case Optimal Datalog to GPUs.
36. Darwiche, A. (2011, July). SDD: A new canonical representation of propositional knowledge bases. In IJCAI Proceedings-International Joint Conference on Artificial Intelligence (Vol. 22, No. 1, p. 819).
37. Darwiche, A. (2004, August). New advances in compiling CNF to decomposable negation normal form. In Proceedings of the 16th European Conference on Artificial Intelligence (pp. 318-322).
38. Yunuo Cen, Zhiwei Zhang, and Xuanyao Fong. Massively Parallel Continuous Local Search for Hybrid SAT Solving on GPUs.
39. Evans, R., & Grefenstette, E. (2018). Learning explanatory rules from noisy data. Journal of Artificial Intelligence Research, 61, 1-64.
40. Jan Wielemaker et al. "SWI-Prolog". In: Theory and Practice of Logic Programming 12.1-2 (2012), pp. 67–96.
41. Somogyi, Z., Henderson, F., & Conway, T. (1996). The execution algorithm of Mercury, an efficient purely declarative logic programming language. The Journal of Logic Programming, 29(1-3), 17-64.
42. Jordan, H., Scholz, B., & Subotić, P. (2016, July). Soufflé: On synthesis of program analyzers. In International Conference on Computer Aided Verification (pp. 422-430). Cham: Springer International Publishing.
43. Gebser, M., Kaminski, R., Kaufmann, B., & Schaub, T. (2019). Multi-shot ASP solving with clingo. Theory and Practice of Logic Programming, 19(1), 27-82.